Biblioteczka
Komputer Świat



NA DVD najlepsze narzędzia programistyczne i pliki szkoleniowe do nauki języka JAVA

JAVA

ZACZNIJ PROGRAMOWAĆ

kurs programowania na prostych przykładach



W KŚ+ dodatkowe pliki szkoleniowe

Z TEJ KSIĄŻKI NAUCZYSZ SIĘ, JAK:

- krok po kroku posługiwać się językiem Java – poznasz składnię i sposób pisania kodu
- zacząć tworzyć gry i programy użytkowe
- pisać aplikacje na Androida
- stworzyć własnego bota



Z TĄ KSIĄŻKĄ E-WYDANIE GRATIS

Poniżej znajduje się płyta z kodem bonusowym dającym dostęp do e-wydania tej książki w serwisie KŚ+ (www.ksplus.pl) oraz pliku ISO z cyfrową wersją płyty do pobrania.

NA PŁYCIE DVD

Płyta zawiera zestaw startowy najlepszych narzędzi dla programistów rozpoczynających swoją przygodę z językiem Java: środowiska programistyczne, edytory kodu źródłowego i pliki szkoleniowe do wskazówek przedstawionych w książce.

Jeżeli brakuje płyty, poinformuj sprzedawcę lub redakcję: redakcja@komputerswiat.pl



JAVA

20 NAJLEPSZYCH NARZĘDZI DLA PROGRAMISTY

PŁYTA JEST DODATKIEM DO KSIĄŻKI KOMPUTER ŚWIAT BIBLIOTECZKA 2|2020

NA PŁYCIE ZNAJDZIESZ KOD DO E-WYDANIA I BONUSÓW W KSPLUS.PL

Kod bonusowy należy zarejestrować w KŚ+ (www.ksplus.pl)

Biblioteczka Komputer Świat

KONRAD JAGACIAK

JAVA

ZACZNIJ PROGRAMOWAĆ

ringier axel springer

AUTOR: Konrad Jagaciak

REDAKTORZY PROWADZĄCY: Rafał Kamiński, Agnieszka Al-Jawahiri

PRZYGOTOWANIE PŁYTY: Mariusz Michalski

PROJEKT OKŁADKI: Robert Dobrzyński

SKŁAD I ŁAMANIE: Mariusz Rybak

KOREKTA: Jolanta Rososińska

WYDAWCA: RINGIER AXEL SPRINGER POLSKA Sp. z o.o.
02-672 Warszawa, ul. Domaniewska 49
tel. 22 7786102
www.ringieraxelspringer.pl

ISBN: 978-83-8091-886-3

Warszawa 2020

DYREKTOR WYDAWNICZY: Paweł Paczuski

BUSINESS PROJECT MANAGER: Paweł Bulwan

DRUK I OPRAWA: Drukarnia im. Adama Półtawskiego, Kielce

EGZEMPLARZE ARCHIWALNE:
www.literia.pl

prenumerata.axel@qg.com

E-WYDANIA: www.ksplus.pl

KONTAKT:
redakcja@komputerswiat.pl

INTERNET: komputerswiat.pl, ksplus.pl

Płyta DVD jest dodatkiem do książki

ringier
axel springer



1 Java w pigułce

„Napisz raz, uruchom wszędzie" – nie sposób innymi słowami rozpocząć książki o języku programowania Java. Angielskie „Write once, run anywhere" (w skrócie WORA) to hasło podkreślające główną zaletę Javy – jej wieloplatformowość

Hasło „Napisz raz, uruchom wszędzie" zostało wymyślone w firmie Sun Microsystems, która stworzyła język programowania Java. I choć Sun Microsystems już nie istnieje – zostało przejęte przez innego giganta IT, firmę Oracle – hasło nadal jest aktualne i na stałe wpisało się w historię informatyki. Pierwsza wersja Javy, stworzona przez grupę roboczą pod kierunkiem Kanadyjczyka Jamesa Goslinga, została wydana w 1995 roku. Przez lata Java zdobyła pozycję jednego z najpopularniejszych języków, a programiści w niej pracujący nie mogą narzekać ani na brak pracy, ani na niskie zarobki.



James Gosling, twórca języka Java

Fot. Peter Campbell/Wikimedia

Maszyna wirtualna

Skąd właściwie bierze się wspomniana wieloplatformowość? Zawdzięczamy ją temu, że Java jest językiem tworzenia programów źródłowych kompilowanych do kodu bajtowego – postaci wykonywanej przez maszynę wirtualną. To właśnie wykorzystanie maszyny wirtualnej sprawia – że kod raz napisany będzie skutecznie działał na sprzęcie o różnej architekturze i z różnymi systemami operacyjnymi (choćby Windows czy Linux). Wymagane jest jednak, by na sprzęcie tym była zainstalowana ta wirtualna maszyna.
Maszyna wirtualna Javy (**Java Virtual Machine**, w skrócie nazywana jest **JVM**) to zestaw aplikacji napisanych na tradycyjne urządzenia i systemy operacyjne. W zależności od potrzeb i liczby dostępnych narzędzi wyróżniane są dwa główne zestawy:

- **JRE – Java Runtime Environment** – zawiera wyłącznie narzędzia niezbędne do uruchomienia aplikacji, czyli środowisko uruchomieniowe;
- **JDK – Java Development Kit** – zawiera również narzędzia dla programistów pozwalające na tworzenie aplikacji na platformę JVM.

Co ważne – określenie wirtualna maszyna Javy nie jest nazwą konkretnego produktu. Dostępna jest specyfikacja pozwalająca różnym producentom oprogramowania na tworzenie własnych maszyn wirtualnych pracujących pod kontrolą różnych środowisk i urządzeń. Firma Oracle Corporation, właściciel znaku towarowego Java, udostępnia swoją maszynę wirtualną. Inne firmy także mogą tworzyć JRE i JDK i używać w swoich produktach znaku Java pod warunkiem, że ściśle przestrzegają oficjalnej specyfikacji i dodatkowych regulacji.



Strona internetowa OpenJDK – openjdk.java.net

W tej książce są przedstawione projekty programistyczne do zrealizowania. Aby je wykonać, nie wystarczy mieć JRE. Niezbędne jest także JDK z narzędziami programistycznymi. Począwszy od Javy 7, wzorcową implementacją JVM jest **OpenJDK** (**DVD-KOD: 019**). Jest to otwarte oprogramowanie i to właśnie jego wykorzystanie opisano w tej książce.

Czy warto uczyć się Javy?

To pytanie można uznać za retoryczne. Java może być dobrym sposobem na początek przygody z programowaniem.
Język ten jest stosunkowo prosty – zawiera jedynie 50 słów kluczowych, a jednocześnie oferuje bogate możliwości dzięki rozbudowanemu zestawowi bibliotek (API), które można wykorzystywać w projektach.
Wspomniana wieloplatformowość Javy przejawia się także tym, że daje nam możliwość pisania aplikacji także na urządzenia mobilne. Znając język Java, można zacząć tworzenie aplikacji na platformę Android. Firma Google udostępnia zestaw SDK (ang. Software Developer Kit) do tworzenia oprogramowania właśnie przy użyciu języka Java.
Ważnym argumentem jest to, że Java to narzędzie potężne – wszechobecne, a co za tym idzie, daje programistom wiele udogodnień, których często brakuje osobom piszącym skrypty w innych językach.

Jak zacząć pracę w Javie

Do rozpoczęcia pracy nad projektami w języku Java może wystarczyć sam pakiet Java Development Kit. OpenJDK nie trzeba instalować, wystarczy wypakować do wybranej lokalizacji archiwum zamieszczone na płycie dołączonej do książki. Warto też wiedzieć, że na stronie **jdk.java.net**, zawsze znajdziemy najnowsze wersje pakietu do pobrania. Teoretycznie, mając JDK, możemy już programować w Javie, wykorzystując do pisania choćby zwykły Notatnik. W praktyce może być to zadanie trudne i niewygodne. Dlatego oprócz JDK dobrze jest wyposażyć się w IDE. W tym wypadku może to być **NetBeans IDE** (**DVD-KOD: 003**).

Aby móc w pełni korzystać z JDK, trzeba jeszcze wykonać kilka czynności. Jeśli tego nie zrobimy, może to uniemożliwić instalację NetBeans IDE.

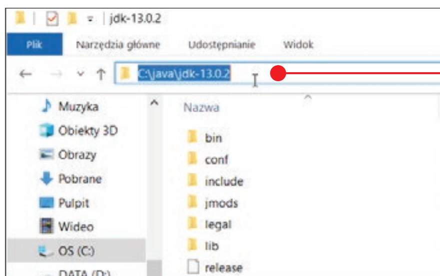


Strona internetowa oprogramowania NetBeans IDE – netbeans.org

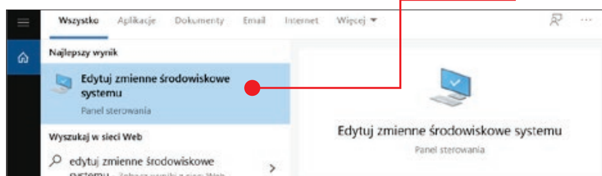


Ustawienie zmiennych środowiskowych

1 Kopiujemy lokalizację folderu, w którym znajduje się rozpakowane OpenJDK.



2 Korzystając z wyszukiwarki systemowej Windows, znajdujemy opcję **Edytuj zmienne środowiskowe systemu**.



3 W otwartym w ten sposób oknie **Właściwości systemu** klikamy na przycisk **Zmienne środowiskowe**.

4 W sekcji **Zmienne systemowe** klikamy na przycisk **Nowa**.

5 W pole **Nazwa** wpisujemy **JAVA_HOME**, a w pole **Wartość** wklejamy skopiowaną wcześniej lokalizację rozpakowanego OpenJDK. Zatwierdzamy dodanie zmiennej przyciskiem **OK**.

6 Po dodaniu zmiennej należy jeszcze edytować zmienną **Path**. Zaznaczamy ją, a następnie klikamy na przycisk **Edytuj**.

7 W oknie edycji zmiennej **Path** klikamy na przycisk **Nowy**. Do pustego paska wklejamy tę samą lokalizację co wcześniej, jednak dopisując na końcu **\bin**, tak by lokalizacja odnosiła się do folderu **bin** w rozpakowanym archiwum OpenJDK. Na koniec przyciskiem **OK** zatwierdzamy edycję zmiennej środowiskowej **Path**.

8 Kolejno przyciskami **OK** zamykamy okna **Zmienne środowiskowe** i **Właściwości systemu**.

Pojęcia warte wyjaśnienia

Zanim przejdziemy do realizacji projektów, warto poznać kilka istotnych pojęć, które będą często pojawiały się w tej książce:

- **Apache Maven** – narzędzie automatyzujące budowę oprogramowania na platformę Java. Podczas budowania potrafi zrobić wszystko z naszym projektem. Jedyne, o czym musimy pamiętać, to stosować odpowiednie nazewnictwo katalogów w projekcie (tak jak wymaga tego Maven). Dalej już sam Maven zatroszczy się o kompilowanie kodu, wykonywanie testów, pilnowanie bibliotek oraz ich wersji, generowanie pliku .jar z naszą aplikacją czy dokumentacji. Zadba także o wiele innych czynności, które bez automatyzacji mogą okazać się czasochłonne, a także powodować występowanie błędów.
- **Dokumentacja programu** – to nazwa odnosząca się do całości dokumentacji, czyli dokumentacji użytkownika i dokumentacji technicznej stworzonej przez twórców programu.
- **Dokumentacja techniczna** – jest tworzona przez twórców na ich własne potrzeby, ale też na potrzeby innych osób chcących modyfikować kod programu. Powinna zawierać dokładny opis poszczególnych elementów kodu, wyjaśniać ich działanie i wykorzystanie.
- **Dokumentacja użytkownika** – to opis programu przeznaczony dla jego użytkownika, w jej skład wchodzą najczęściej pliki pomocy i instrukcja obsługi.
- **Instrukcja warunkowa** – element języków programowania, który uzależnia wykonywanie wybranych przez programistę instrukcji od tego, czy przedstawione wyrażenie logiczne jest prawdą, czy też fałszem.
- **Klasa** – termin odnoszący się do paradygmatu programowania obiektowego. Jest to definicja (lub częściowa definicja) obiektów. Klas używa się bardzo często do odzwierciedlenia świata rzeczywistego w kodzie programu.
- **Metoda** – podprogram składowy klasy. Odpowiada za działanie konkretnych elementów klasy. Na przykład dla klasy **Piłkarz** metodami mogą być wykonywane przez Piłkarza czynności, takie jak **bieg**, **strzał**, **podanie** czy **wślizg**.
- **Obiekt** – instancja klasy. Na przykład, jeśli mamy klasę **Piłkarz**, obiektami będą poszczególni piłkarze.
- **Pętla** – jedna z podstawowych konstrukcji programistycznych. Pętle umożliwiają cykliczne wykonywanie ciągu instrukcji. W zależności od rodzaju pętli cykl ten jest wykonywany określoną liczbę razy do momentu zajścia pewnych warunków dla każdego elementu kolekcji lub w nieskończoność.
- **Swing** – biblioteka graficzna Javy. To dzięki niej możemy korzystać z takich elementów, jak przyciski, pola tekstowe czy panele, a także inne elementy wyglądu aplikacji.
- **Testy jednostkowe** – metoda sprawdzania poprawności działania programu poprzez sprawdzanie jego poszczególnych elementów (jednostek), na przykład metod, obiektów, procedur, zależnie od typu języka.
- **Zmienna** – w uproszczeniu jest to konstrukcja, która pod określoną nazwą pozwala na przechowywanie wartości.

Java – podstawy składni

Java jest językiem zorientowanym obiektowo. To oznacza, że podczas pisania będziemy tworzyć tak zwane klasy. Są one definicjami dla obiektów, które w zasadzie powinny odzwierciedlać świat zewnętrzny. Również na poszczególne elementy programu będziemy patrzeć przez pryzmat klas. I tak obiektem jest też na przykład okno programu.

O tym, jak tworzyć klasy, dowiemy się w dalszej części książki. Podczas tworzenia klas nie obejdzie się na przykład bez konieczności deklaracji zmiennych. Te w skrócie możemy określić jako parę, w skład której wchodzi nazwa i wartość przechowywana pod konkretną nazwą.
Java jest jednak językiem silnie typowanym – to oznacza, że każda zmienna czy pole

TYPY DANYCH

Liczby całkowite:

- **byte** – nazwa tego typu danych pochodzi od „bajta". Przyjmuje się, że bajt to w jednostkach informacji pamięci komputerowej 8 bitów (bit to najmniejsza jednostka danych). Na zmienną typu byte mówi się zatem, że jest 8-bitowa. Liczba bitów wpływa na zakres wartości możliwych do przechowania w zmiennej. I tak w przypadku tego typu danych mogą tam się mieścić liczby całkowite od -128 do 127.
- **short** – w zmiennych tego typu możemy przechowywać tego samego rodzaju dane, co w zmiennych typu byte, jednak inny jest ich zakres. Ten typ danych może przechowywać wartości na 16 bitach. Co daje nam już zakres liczb całkowitych od -32 768 do 32 767.
- **int** – jest najszerzej wykorzystywanym typem danych do przechowywania liczb całkowitych. Może umieścić wartość na 32 bitach, co oznacza, że liczba całkowita musi mieścić się w zakresie od -2147483648 do 2147483647. Taki zakres zdaniem programistów bardzo często jest wystarczający względem wymagań dotyczących wartości, jakie mogą być lokowane w zmiennej.
- **long** – do przechowywania liczb całkowitych może wykorzystać aż 64 bity.

Liczby zmiennoprzecinkowe:

- **float** – może przechowywać liczby na 32 bitach. Wielkość zmiennej zmiennoprzecinkowej wpływa też na precyzję. Jeśli mamy ułamek z wieloma cyframi po przecinku – końcowa część, która nie zmieści się w zmiennej, jest obcinana.
- **double** – może przechowywać liczby na 64 bitach – cechuje się zatem większą dokładnością od typu float.

UWAGA! Przypisując wartości do zmiennych typów zmiennoprzecinkowych, do oddzielenia części całkowitej od części ułamkowej nie korzystamy z przecinka, ale z kropki!

Inne:

- **char** – typ danych służący do znaków, czyli między innymi liter oraz cyfr. Ten typ zaliczany jest do typów prostych. Większe znaczenie ma obiektowy typ danych **string**, który jest ciągiem takich znaków. To w nim będziemy mogli zapisywać całe słowa, zdania czy akapity.
- **boolean** – najmniejszy możliwy typ danych. Mieści się na jednym bicie. Może przechowywać tylko jedną z dwóch wartości – true lub false.

(polem nazywamy zmienną działającą w obszarze klasy) oprócz nazwy i wartości musi mieć jeszcze swój zadeklarowany typ.
I tak, jeśli pod nazwą **punkty** chcemy przechowywać w grze liczbę zdobytych przez gracza punktów, to powinniśmy zastanowić się, jaka może to być liczba.

To dlatego, że typ zmiennych odpowiada za rodzaj i zakres wartości, jakie może ta zmienna przechowywać. Innych typów zmiennych używamy do przechowywania liczb całkowitych, a innych do przechowywania ułamków (te nazywamy liczbami zmiennoprzecinkowymi).

```
package com.mycompany.mavenexample; A

import java.awt.Graphics;
B import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
               C          D                                 E
public class jPanel1 extends javax.swing.JPanel implements ActionListener {

  F int pal_h,pal_w,pal_x,pal_y;
    Timer timer;
    Graphics gr;

  G public jPanel1() {
        initComponents();
        pal_h = 10;
        pal_w = 50;
        pal_x = NewJFrame.WIDTH/2 - pal_w/2;
        pal_y = NewJFrame.HEIGHT - pal_h;
        timer = new Timer(150, this);
        timer.start();
    }
  H @Override
    public void actionPerformed(ActionEvent e) {
        repaint();
    }
    private void initComponents() {

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(this);
        this.setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 400, Short.MAX_VALUE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 300, Short.MAX_VALUE)
        );
    }

    @Override
    public void paintComponents(Graphics g)
    {
        g.drawRect(pal_x, pal_y, pal_w, pal_h);
    }
}
```

Typy zmiennych różnią się między sobą także rozmiarem. To znaczy, że mogą przechowywać wartości na różnej liczbie bitów w pamięci. Nierozsądne deklarowanie zmiennych może prowadzić do dużego zapotrzebowania programu na pamięć operacyjną. Dlatego dobierając typ zmiennych, analizujemy najpierw, jakie wartości może ona przyjąć i w jakim zakresie się one mieszczą, aby skorzystać z najmniejszego możliwego typu, wystarczającego do przechowania potrzebnej wartości.

Podstawy składni języka Java

Podstawy składni języka Java omówimy na przykładowym skrypcie (patrz ilustracja na stronie obok).

Już na pierwszy rzut oka zwracają uwagę nawiasy klamrowe **{ }**. W Javie odpowiadają one za wydzielanie fragmentów kodu.

To, co znajduje się wewnątrz nawiasu klamrowego, może być treścią – klasy, metody czy pętli lub innej konstrukcji.

CO TO JEST DZIEDZICZENIE

Dziedziczenie jest jednym z fundamentów programowania obiektowego. Jego ideę najlepiej wytłumaczyć na przykładzie.

Załóżmy, że mamy klasę **Siatkarz**. Siatkarze grający na różnych pozycjach na boisku mają wiele wspólnych zachowań i właściwości.

Są jednak także cechy, którymi się różnią, na przykład zawodnik libero nie atakuje.

Wszystkie aspekty wspólne dla siatkarskich pozycji możemy opisać w klasie **Siatkarz**. A tworząc w grze zawodników na różnych pozycjach, możemy tworzyć klasy dziedziczące po klasie **Siatkarz**.

W ten sposób zaoszczędzimy sobie pracy – elementy wspólne klas trafiają do nich na zasadzie dziedziczenia, bez konieczności pisania ich w każdej z klas.

Niektóre konstrukcje zawierają się wewnątrz innych, stąd często u dołu skryptów widzimy wiele zamknięć nawiasów – każdy zamyka inną część skryptu.

Przyjrzyjmy się kilku linijkom tego kodu, aby poznać ich działanie:

A **package** – pozwala określić przestrzeń nazw projektu. Mówiąc ogólnie – jest zbiorem między innymi klas i interfejsów, z których jest zbudowany projekt.

B **import** – pozwala dodać do projektu zawartość innych przestrzeni nazw.

C słowo **class** pozwala zadeklarować klasę, w tym przypadku klasę o nazwie **jPanel1**.

D słowo **extends** oznacza, że nasza klasa będzie dziedziczyć po innej klasie – **javax.swing.JPanel**.

E **implements** – pozwala na dodanie interfejsu o nazwie **ActionListener**, po którym klasa również może dziedziczyć. Interfejs nazywany jest często „spisem treści" dla klasy. Interfejs w kontekście programowania w języku Java to zestaw metod bez ich implementacji (bez kodu definiującego zachowanie każdej metody).

F **deklaracja pól klasy**, czyli mówiąc w skrócie – zmiennych, które są dostępne dla wszystkich metod tej klasy.

G specjalna metoda klasy nazywana **konstruktorem**. Dzięki niej możliwe jest tworzenie obiektów klasy. Najczęściej to w konstruktorze pola klasy dostają swoje wartości.

H **@Override** oznacza, że kolejna opisana metoda zostaje nadpisana. To znaczy: treść metody odziedziczona po klasie, po której zrobiliśmy dziedziczenie, zostaje zastąpiona nową treścią.

2 Jak pracować w Javie

Do tworzenia własnych programów w Javie wystarczy JDK i prosty edytor tekstu. Nie jest to jednak jedyna możliwość. W tym rozdziale zobaczymy też, że do pracy w Javie można wykorzystać także dodatkowe oprogramowanie – tak zwane IDE

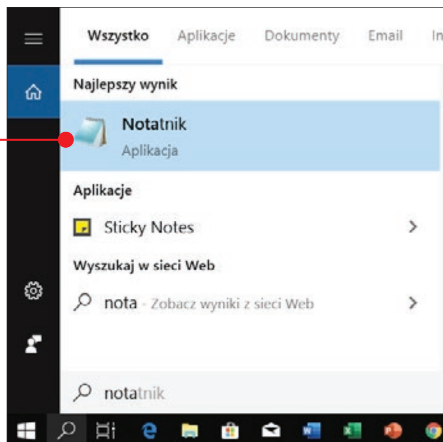
Realizując wskazówki przedstawione w tym rozdziale, stworzymy swoje pierwsze skrypty w Javie. Kiedy zaopatrzymy swój komputer w JDK, wystarczy użycie najprostszego edytora tekstu, nawet Notatnika, by móc tworzyć swoje pierwsze programy w Javie. Pozwoli to także lepiej zrozumieć podstawy języka. A następnie zapoznamy się z działaniem IDE – zintegrowanego środowiska programistycznego.

IDE

IDE, czyli zintegrowane środowisko programistyczne, to program lub zespół programów służących do tworzenia, modyfikowania, testowania i konserwacji oprogramowania. IDE jest bardzo ważne – chociaż nie jest niezbędne, by rozpocząć pracę w Javie, to bardzo ją usprawnia.
Dlatego w dalszej części tego rozdziału i w kolejnych opisano wykorzystanie **Apache NetBeans IDE** (DVD-KOD: 003). Warto wiedzieć, że podobną funkcjonalność oferują też na przykład **Eclipse IDE** (DVD-KOD: 006) czy **IntelliJ** (DVD-KOD: 013), które także znajdują się na płycie dołączonej do książki.

Hello World

Hello World (Witaj, świecie) to klasyczny program, którego celem jest wypisanie na standardowym wyjściu napisu „Hello World!" (lub innego dowolnego, prostego komunikatu). Jego przeznaczeniem jest jedynie demonstracja języka czy też środowiska, w jakim został napisany. Nazwą tą określa się też różne inne proste programy, których jedynym zadaniem jest pokazywanie działania języka programowania. Początkujący programiści, ucząc się nowego języka lub w ogóle zaczynając naukę programowania, często stawiają sobie jako pierwsze zadanie samodzielne napisanie programu Hello World. My również tak zrobimy:



1 Otwieramy **Notatnik** systemowy, aby rozpocząć pisanie skryptu.

2 Pierwsza linijka – **public class HelloWorld{** A – zawiera deklarację publicznej klasy o nazwie **HelloWorld**. Kończy się otwarciem nawiasu klamrowego. Wszystko, co napiszemy przed jego zamknięciem, będzie treścią naszej klasy.

```
public class HelloWorld{ A
        public static void main(String[] args){ B
                System.out.println("Hello World"); C
        }
}
```

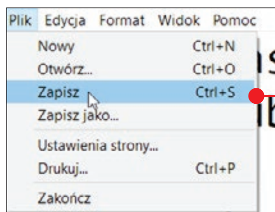
3 Linijka druga – **public static void main(String[] args){** B – zawiera deklarację głównej metody aplikacji: **main**.
Ta metoda jest uruchamiana jako pierwsza w aplikacji i dopiero w niej wywoływane są inne metody.
Po nazwie metody znajduje się deklaracja przyjmowanych argumentów, w tym wypadku są to również argumenty wejściowe całej aplikacji. Linijka kończy się otwarciem nawiasu klamrowego – to oznacza, że dalsza część skryptu, aż do zamknięcia tego nawiasu, będzie treścią metody **main**.

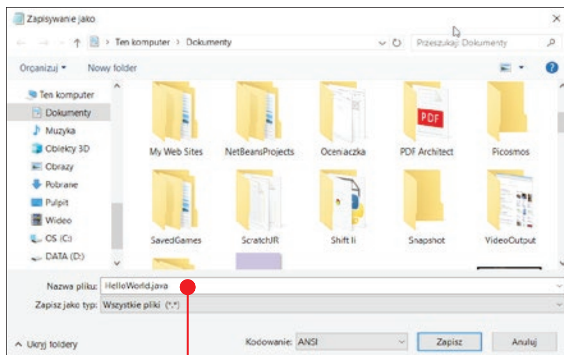
4 Trzecia linia kodu – **System.out.println ("Hello World");** C – to właściwe ciało naszej aplikacji. Jest to wywołanie metody wyświetlającej tekst na konsoli. Tym razem nie otwieramy klamrowego nawiasu, ponieważ nic tu nie definiujemy, tylko wywołujemy konkretne polecenie. Wywołanie metody kończy się znakiem średnika, można go porównać do kropki kończącej zdanie. W kolejnych liniach moglibyśmy wypisywać kolejne polecenia – wykonywałyby się one w kolejności, w jakiej byśmy je ułożyli. Podany do wyświetlenia w konsoli tekst został wpisany w cudzysłowie, można w nim wpisać dowolną treść, jaką chcemy wyświetlić.

5 Kolejna linijka kodu zawiera zamknięcie nawiasu klamrowego. Zamyka ten nawias, który został ostatnio otwarty, czyli kończy treść metody **main**.

6 Drugie zamknięcie, w kolejnej linii skryptu, odnosi się do najbliższego otwartego nawiasu klamrowego, który nie został jeszcze zamknięty. Zamyka zatem definicję klasy.

7 Napisany w ten sposób skrypt powinniśmy zapisać do odpowiedniego pliku. Z menu **Plik** wybieramy opcję **Zapisz** ●.



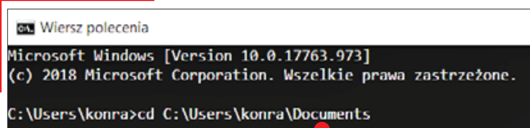


8 W oknie dialogowym wybieramy lokalizację, w której zostanie zapisany plik. Wpisujemy też jego nazwę – **HelloWorld.java** ●, a w polu **Zapisz jako typ** wybieramy opcję **Wszystkie pliki**.

Kompilacja

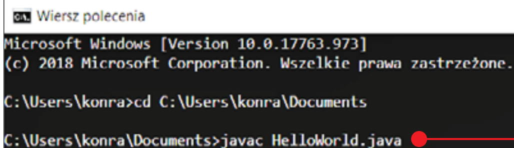
Tak napisany skrypt sam się jeszcze nie wykona. Trzeba dokonać jego kompilacji, by wytworzyć kolejny plik, i dopiero jego uruchomienie spowoduje wykonanie skryptu. W tym celu skorzystamy z **Wiersza polecenia** ● – znajdujemy go i uruchamiamy.

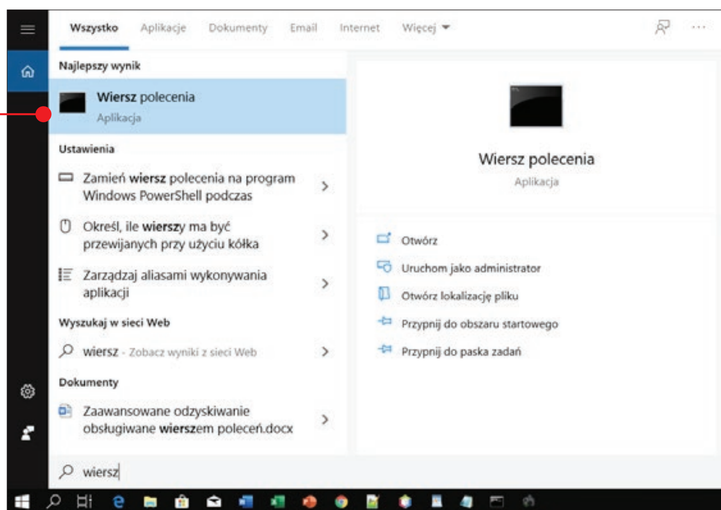
1 Przez polecenie **cd** przechodzimy do lokalizacji, w której zapisany został plik HelloWorld.java, wpisując ją po **cd** ●.

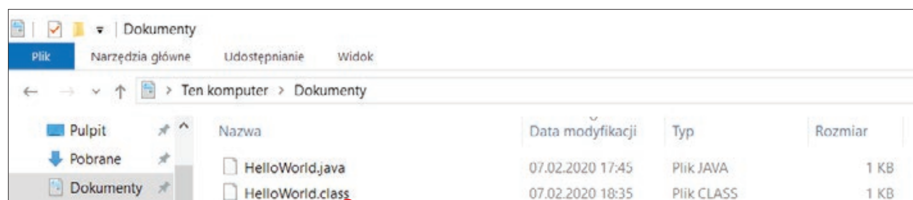


2 Gdy będziemy już w lokalizacji pliku, wpisujemy **javac HelloWorld.java** ●.

3 Sprawdzamy folder, w którym znajduje się plik **HelloWorld.java** – użyte w poprzednim kroku polecenie powinno wyge-







nerować plik **HelloWorld.class**. Zawiera on instrukcje wykonywane przez maszynę wirtualną Javy.

```
C:\Users\konra\Documents>javac HelloWorld.java

C:\Users\konra\Documents>java HelloWorld
Hello World

C:\Users\konra\Documents>
```

4 W Wierszu polecenia wpisujemy teraz komendę **java HelloWorld**, by uruchomić plik.

5 Wiersz polecenia w odpowiedzi wykona skrypt – czyli wyświetli tekst **Hello World**.

6 W ten sposób powstał nasz pierwszy – bardzo prosty – program w języku Java!

Gra liczbowa: mnożenie wylosowanych liczb

Piszemy skrypt

W ten sam sposób możemy stworzyć bardziej rozbudowany skrypt, który będzie prostą grą liczbową, jednak jego napisanie pozwoli na poznanie kolejnych instrukcji. Gra będzie polegała na podaniu poprawnego wyniku mnożenia dwóch liczb wylosowanych przez program. Znowu zaczynamy od uruchomienia Notatnika, by móc pisać nowy skrypt.

1 Tak jak w pierwszym projekcie – rozpoczniemy od klasy. Wpisujemy **public class Gra{**. Co ważne, nazwa klasy – w tym przypadku **Gra** – musi być też nazwą pliku. Podczas zapisywania pliku musimy nazwać go **Gra.java**.

```
public class Gra{
```

```
public class Gra{
        public static void main(String[] args){
```

2 W kolejnej linii kodu powinniśmy utworzyć metodę **main**, czyli główną metodę klasy, w której zawrzemy instrukcje do wykonania po uruchomieniu programu.

3 Trzecia linia kodu, jaką napiszemy, będzie instrukcją, która wykona się jako pierwsza w naszym programie. Będzie to utworzenie generatora liczb pseudolosowych (w programowaniu nie ma liczb losowych – mamy liczby pseudolosowe, których dobór zależy od pewnych warunków, tak by ich wartość była możliwie trudna do odgadnięcia). Generator będzie obiektem klasy **Random**. Tworząc obiekt jakiejś klasy, najpierw podajemy nazwę tej klasy – tu **Random**, potem nazwę obiektu – tu **generator**. Dalej po znaku równości uruchamiamy konstruktor, pisząc **new Random()**. Linię kończymy średnikiem. Całość powinna mieć zatem formę: **Random generator = new Random();**.

```
public class Gra{
        public static void main(String[] args){
                Random generator = new Random();
```

```
import java.util.Random;

public class Gra{
        public static void main(String[] args){
                Random generator = new Random();
```

4 Klasa **Random** nie jest bezpośrednio rozpoznawana przez kompilator. Trzeba do skryptu załączyć odpowiednią bibliotekę, w której znajduje się klasa **Random**. Biblioteki do skryptu załączamy poprzez polecenie **import** – na samym początku skryptu. Klasa **Random** znajduje się w **java.util.Random**. Piszemy zatem **import java.util.Random;**. Pamiętajmy: import bibliotek umieszcza się na początku skryptu – przed definicją klas.

5 Teraz już bez obaw, że polecenie nie zostanie rozpoznane, możemy korzystać z obiektu **generator**. Pozwala on na generowanie wartości różnych typów. Załóżmy, że chcemy mieć liczbę całkowitą z zakresu od 0 do 200. Aby ją przechować, potrzebna będzie zmienna. Tworząc nową zmienną, najpierw określamy jej typ. Powinien to być typ odpowiadający za liczby całkowite, czyli byte, short, int lub long. Nie skorzystamy jednak na pewno z typu byte, bo możliwa do „wylosowania" wartość przekroczyłaby zakres tego typu zmiennej. 200 mieści się w zakresie pozostałych typów. Choć maksymalna wartość zmieściłaby się w zmiennej typu **short**, użycie innego typu, na przykład **int**, nie będzie błędem. Po nazwie typu należy podać nazwę zmiennej, na przykład **x**. Dalej dajemy znak równości, by przypisać do zmiennej wartość. Będzie ona nadana poprzez generator – a konkretnie poprzez wywołanie jednej z jego metod. Do generowania liczb w typie int służy polecenie **nextInt()** – gdzie w nawiasie możemy podać maksymalną możliwą wartość do „wylosowania". Cała linia kodu powinna w tym wypadku mieć postać: **int x = generator.nextInt(200);** A.

```
import java.util.Random;

public class Gra{
        public static void main(String[] args){
                Random generator = new Random();
                int x = generator.nextInt(200); A
                int y = generator.nextInt(200); B
```

6 Ponieważ w naszym projekcie potrzebujemy dwóch liczb, aby wykonać ich mnożenie, tworzymy drugą zmienną, **y** B – i jej też nadajemy wartość poprzez **generator**.

7 Kolejna linia kodu powinna informować użytkownika naszej gry o tym, co powinien zrobić. Musimy zatem wpisać w konsoli informację. Robimy to za pomocą znanego nam już polecenia **System.out.println()**. Tym razem jednak już nie będziemy wpisywać "Hello World", ale "Podaj wynik mnożenia". Użytkownik powinien jednak wiedzieć, jakie liczby ma pomnożyć. Jeśli do "Podaj wynik mnożenia" dodamy **x**, to otrzymamy ciąg znaków składający się z tekstu w cudzysłowie i wartości zmiennej **x**. Całe zdanie, w którym podajemy działanie do rozwiązania, powinno mieć zatem formę: **"Podaj wynik mnożenia:"+x+"*"+y**.

```
Random generator = new Random();
int x = generator.nextInt(200);
int y = generator.nextInt(200);
System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
```

8 Dalej użytkownik powinien mieć możliwość podania odpowiedzi. By móc odczytywać dane od użytkownika, będzie nam potrzebny obiekt klasy **Scanner**, który stworzymy poleceniem **Scanner scan = new Scanner(System.in);**.

```
Random generator = new Random();
int x = generator.nextInt(200);
int y = generator.nextInt(200);
System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
Scanner scan = new Scanner(System.in);
```

```
import java.util.Scanner;
import java.util.Random;

public class Gra{
        public static void main(String[] args){
                Random generator = new Random();
                int x = generator.nextInt(200);
                int y = generator.nextInt(200);
                System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
                Scanner scan = new Scanner(System.in);
```

9 **Scanner**, podobnie jak **Random**, nie zostanie rozpoznany przez kompilator, jeśli nie dodamy odpowiedniej biblioteki. Klasa **Scanner** znajduje się w **java.util.Scanner** i by ją importować, przed definicją klasy trzeba umieścić linijkę **import java.util.Scanner;** W tym przypadku nie ma znaczenia, która z bibliotek będzie importowana jako pierwsza (można to zapisać zarówno przed linijką **import java.utilRandom**, jak i po niej).

10 To, co poda użytkownik, będziemy musieli przechować w zmiennej. Wynik mnożenia dwóch liczb całkowitych będzie liczbą całkowitą. Możemy zatem użyć typu int. Tworzymy zmienną o nazwie **odp**, typu **int**. Jako jej wartość przypisujemy wywołanie metody **nextInt()** dla obiektu klasy **Scanner**.

```
Random generator = new Random();
int x = generator.nextInt(200);
int y = generator.nextInt(200);
System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
Scanner scan = new Scanner(System.in);
int odp = scan.nextInt();
```

11 Kolejnym krokiem będzie porównanie podanej przez użytkownika liczby z wynikiem mnożenia **x** i **y**. Do takich porównań używa się **instrukcji warunkowej** – czyli **if**. Po tym słowie otwieramy zwykły nawias, w którym należy wpisać wartość wyrażenia, którego poprawność chcemy sprawdzić. W naszym wypadku będzie to **odp==x*y**. Do porównań używamy dwóch znaków równości (jeden znak równości oznacza przypisanie). Po nawiasie z wpisanym warunkiem otwieramy nawias klamrowy – to co napiszemy do czasu jego zamknięcia, będzie się wykonywać, gdy podany przez nas warunek zostanie spełniony (czyli gdy użytkownik udzieli poprawnej odpowiedzi).

```
Random generator = new Random();
int x = generator.nextInt(200);
int y = generator.nextInt(200);
System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
Scanner scan = new Scanner(System.in);
int odp = scan.nextInt();
if (odp==x*y){
```

12 Jeśli użytkownik udzieli poprawnej odpowiedzi, powinniśmy go o tym poinformować – zrobimy to poleceniem **System.out.println(”Dobrze”);**.

```
if (odp==x*y){
        System.out.println("Dobrze");
```

```
if (odp==x*y){
        System.out.println("Dobrze");
}
```

13 Jeśli nie chcemy już wykonywać innych instrukcji, gdy warunek został spełniony, możemy zamknąć nawias klamrowy.

14 Jeśli chcemy, by coś wykonało się, gdy warunek podany w instrukcji warunkowej nie zostanie spełniony, musimy dodać sekcję **else**, której treść (instrukcje do wykonania) także wpiszemy w nawias klamrowy.

```
if (odp==x*y){
        System.out.println("Dobrze");
}
else {
```

15 Analogicznie jak w wypadku udzielenia poprawnej odpowiedzi, gdy użyt-

kownik udzieli błędnej odpowiedzi, również go o tym poinformujemy, a możemy to zrobić poleceniem **System.out.println("Źle");**.

```
if (odp==x*y){
        System.out.println("Dobrze");
}
else {
        System.out.println("Źle");
```

16 Jeżeli nie chcemy wykonywać więcej instrukcji, powinniśmy zamknąć sekcję **else** nawiasem klamrowym.

```
if (odp==x*y){
        System.out.println("Dobrze");
}
else {
        System.out.println("Źle");
}
```

17 Chcąc zakończyć definicję głównej metody naszej klasy, także zamykamy nawias klamrowy.

18 Podobnie jest z zakończeniem definicji całej klasy – też zamykamy ją nawiasem klamrowym **A**.

Kompilacja i uruchomienie

Podobnie, jak w wypadku Hello World, wykorzystamy Wiersz polecenia do kompilacji i uruchomienia programu.

1 Poprzez polecenie **cd** przenosimy się do lokalizacji pliku ze skryptem.

```
Wiersz polecenia
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. Wszelkie prawa

C:\Users\konra>cd C:\Users\konra\Documents
```

2 Uruchamiamy kompilator javac dla pliku **Gra.java**.

```
C:\Users\konra>cd C:\Users\konra\Documents

C:\Users\konra\Documents>javac Gra.java
```

```
public static void main(String[] args){
        Random generator = new Random();
        int x = generator.nextInt(200);
        int y = generator.nextInt(200);
        System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
        Scanner scan = new Scanner(System.in);
        int odp = scan.nextInt();
        if (odp==x*y){
                System.out.println("Dobrze");
        }
        else {
                System.out.println("Źle");
        }
}
```

```
public class Gra{
        public static void main(String[] args){
                Random generator = new Random();
                int x = generator.nextInt(200);
                int y = generator.nextInt(200);
                System.out.println("Podaj wynik mnożenia:"+x+"*"+y);
                Scanner scan = new Scanner(System.in);
                int odp = scan.nextInt();
                if (odp==x*y){
                        System.out.println("Dobrze");
                }
                else {
                        System.out.println("Źle");
                }
        }
}
```
A

```
Wiersz polecenia
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\konra>cd C:\Users\konra\Documents

C:\Users\konra\Documents>javac Gra.java

C:\Users\konra\Documents>java Gra
```

3 Następnie poleceniem **java Gra** włączamy program.

4 Postępujemy zgodnie z informacjami podanymi przez program i podajemy wynik. Gra powinna wskazać, czy podany wynik jest prawidłowy, czy nie.

```
C:\Users\konra\Documents>javac Gra.java

C:\Users\konra\Documents>java Gra
Podaj wynik mnożenia:50*8
400
Dobrze

C:\Users\konra\Documents>
```

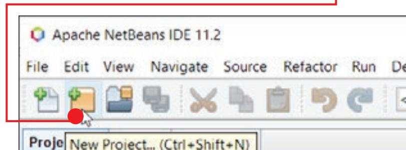
Wykorzystanie zintegrowanego środowiska programistycznego

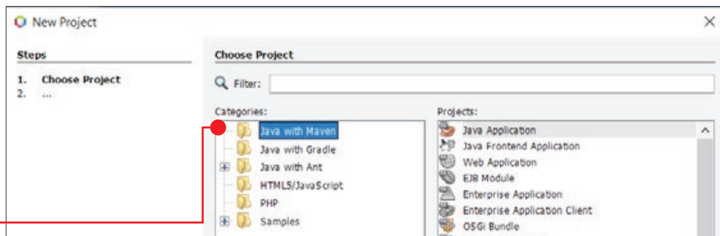
W czym wykorzystanie zintegrowanego środowiska programistycznego powinno ułatwić pracę?
Zaprezentowane skrypty to programy działające w formie tekstowej. Jeśli myślimy o tworzeniu gier, na pewno chcielibyśmy, by miały one okno graficzne. Zintegrowane środowiska programistyczne pozwalają na przykład na proste generowanie takiego okna poprzez moduł graficzny, w którym wygląd programu możemy tak naprawdę „wyklikać” – ręcznie umieścić w jego oknie choćby przyciski czy inne ważne elementy. Gdy brakuje nam takiego narzędzia, wszystko trzeba napisać, a to może być bardzo czasochłonne.
Innym ważnym aspektem wykorzystywania IDE są biblioteki. Korzystając z klas, do których niezbędny jest import bibliotek, musimy wiedzieć, skąd je brać. IDE potrafi samo podpowiedzieć, skąd możemy zaimportować bibliotekę.
Kolejna zaleta zintegrowanych środowisk programistycznych to możliwość korzystania z podpowiedzi podczas pisania skryptu. Programy te w trakcie pisania sugerują możliwe dokończenie instrukcji, co także przyspiesza tworzenie. Co więcej, napisane już skrypty są często pokolorowane, a dzięki temu – bardziej czytelne.
To tylko niektóre z zalet IDE. Jest ich znacznie więcej. W dalszej części tej książki zaprezentowane zostanie wykorzystanie programu **Apache NetBeans IDE**.

Tworzenie nowego projektu

Uruchamiamy **Apache NetBeans IDE** (**DVD-KOD: 003**).

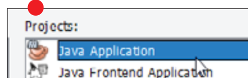
1 By utworzyć nowy projekt, klikamy u góry okna na drugi przycisk od lewej.

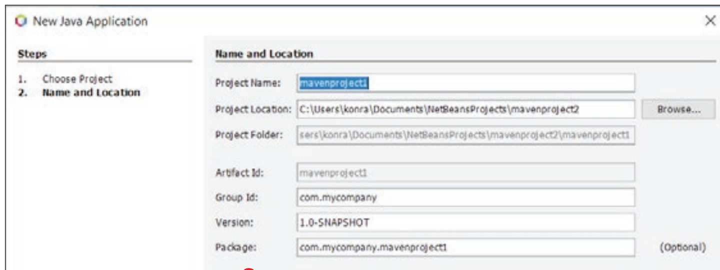




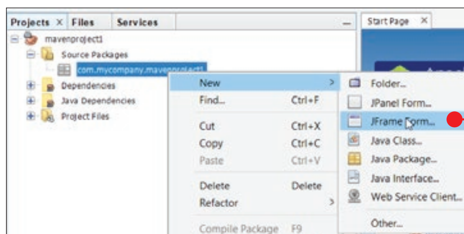
2 W nowym oknie należy wybrać kategorię tworzonego projektu. My na początek będziemy korzystać z kategorii **Java with Maven**.

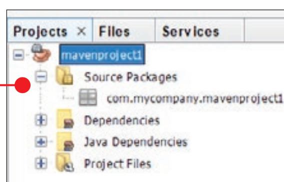
3 W sekcji obok wybieramy typ projektu **Java Application**, po czym przyciskiem **Next** przechodzimy dalej.





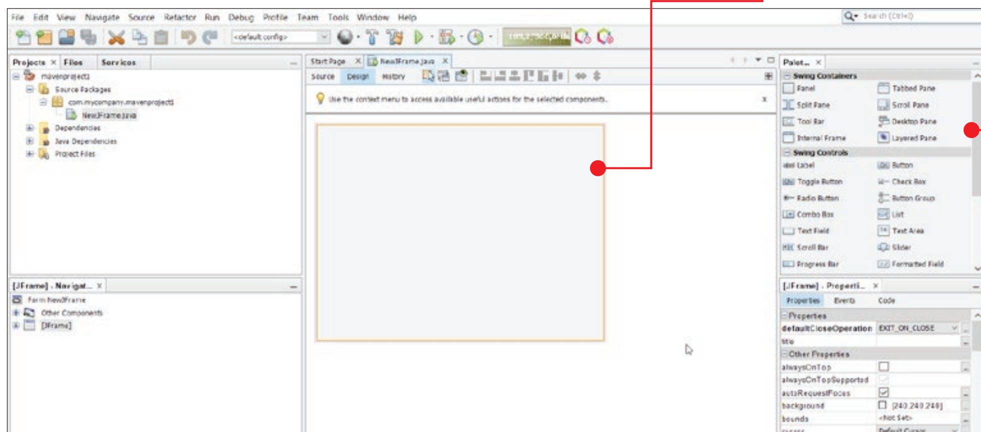
4 W kolejnym kroku możemy podać między innymi nazwę i lokalizację naszego projektu. By został utworzony, klikamy na **Finish**.

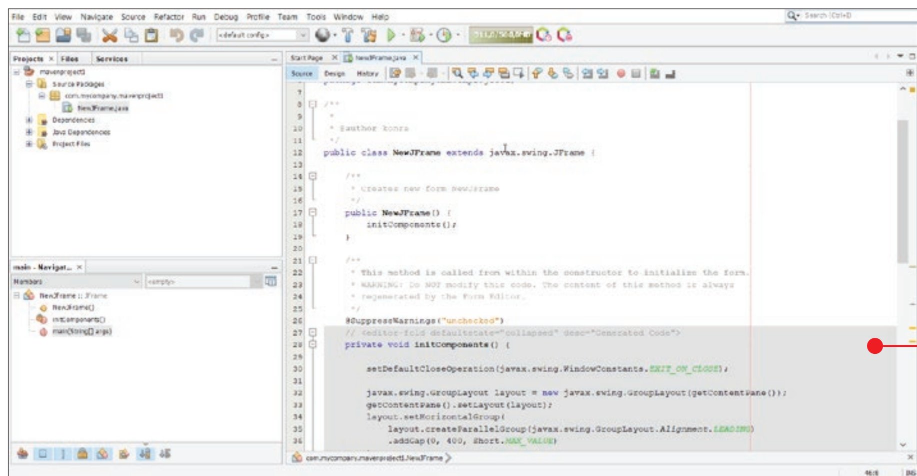


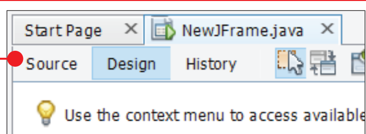


5 Gdy projekt zostanie utworzony, możemy rozwinąć jego zawartość – jak na ilustracji.

6 Klikamy prawym przyciskiem myszy na nazwę paczki utworzonego projektu. Z menu kontekstowego rozwijamy pozycję **New** i wybieramy **JFrame Form**. Spowoduje to dodanie do naszego projektu formatki, czyli okna graficznego.







7 Jego elementy możemy tworzyć, przeciągając kontrolki na schemat okna z palety po prawej stronie okna programu NetBeans. Możemy też przełączyć się do trybu tekstowego i zobaczyć skrypt naszego programu.

Gdy klikniemy na przycisk **Source**, program przeniesie nas do skryptu, który wygenerował się sam i odpowiada za utworzenie okna. Sam wygenerowany skrypt zajmuje ponad 80 linii kodu – co już pokazuje, jak pomocne jest wykorzystanie IDE.
By lepiej zapoznać się z obsługą NetBeans IDE, wykonajmy wskazówki z kolejnego rozdziału. Dzięki temu poznamy tajniki nie tylko Javy, ale i stosowania IDE.

KOMENTARZE

W skrypcie nie wszystkie teksty mają równorzędne znaczenie dla działania programu. Brak niektórych z nich nie wpłynąłby na program. Część wygenerowanej treści to **komentarze**. Nie są one brane pod uwagę podczas wykonywania skryptów. Komentarze wykorzystywane są przez programistów do zapisywania informacji o fragmentach kodu – na przyszłość dla siebie, gdyby powrócili do edycji skryptu po przerwie i nie pamiętali przeznaczenia poszczególnych nazw, a także dla innych programistów. Komentarz pozwala też oznaczyć fragment kodu w ten sposób, by zrezygnować z jego działania, jednocześnie nie usuwając napisanych już skryptów. Można wtedy, po usunięciu oznaczeń komentarza, łatwo przywrócić go do użycia. Są narzędzia, które na

```
robot.setAutoDelay(500);
//tu znajduje się komentarz
robot.keyPress(KeyEvent.VK_J);
robot.keyPress(KeyEvent.VK_E);
robot.keyPress(KeyEvent.VK_S);
robot.keyPress(KeyEvent.VK_T);
robot.keyPress(KeyEvent.VK_E);
/* Tutaj także
znajduje się komentarz, jednak
```

podstawie komentarzy mogą generować dokumentację. Czym jest, jakie znaczenie ma dokumentacja i w jaki sposób tworzyć komentarze, by pozwalały na jej generowanie, przeczytamy w dalszej części książki. Zwykły komentarz możemy dodać, używając dwóch znaków **//** – wtedy wszystko, co jest napisane po tym oznaczeniu do końca linii, jest uważane za komentarz i nie ma wpływu na działanie programu. By stworzyć komentarz na kilka linijek, zaczynamy go od **/***, a w miejscu jego zakończenia piszemy ***/**.

3 Java w praktyce

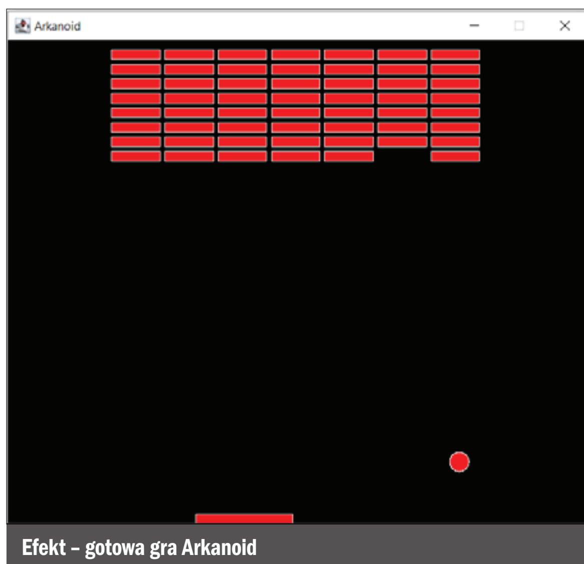
Tajniki języków programowania najlepiej poznawać, tworząc projekty. W tym rozdziale znajdują się instrukcje, według których z wykorzystaniem programu NetBeans IDE nauczymy się tworzyć grę z oknem graficznym

Arkanoid

Grą graficzną, jaką stworzymy, będzie projekt Arkanoid. Rozgrywka będzie polegała na takim sterowaniu paletką, by odbijać nią piłkę, zbijając znajdujące się ponad paletką bloczki (cegły).

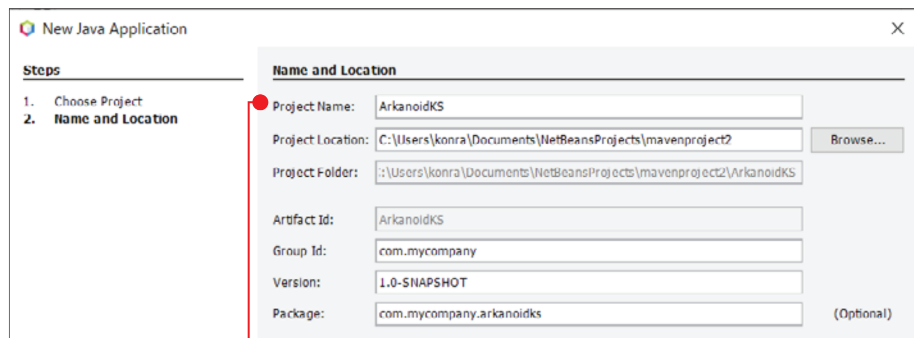
Pierwowzór naszej gry został stworzony i wydany przez firmę Taito w 1986 roku na automaty do gry. Do dziś jest to popularny rodzaj gier, który nieco ewoluował, a kolejne wersje są coraz bardziej rozbudowane.

My na początek stworzymy bazę do gry, którą poznając tajniki języka Java, będzie można z czasem samodzielnie rozbudowywać.



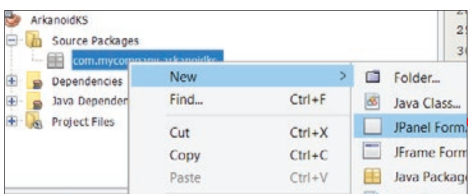
Efekt – gotowa gra Arkanoid

Tworzenie projektu

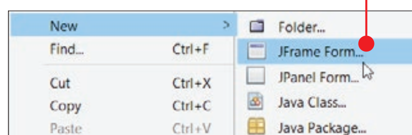


1 Uruchamiamy program **Apache NetBeans IDE** (**DVD-KOD: 003**) i tworzymy nowy projekt, tak jak zostało to opisane w punktach **1**–**5** na końcu poprzedniego rozdziału. Podczas tworzenia projektu nadajemy mu nazwę odpowiadającą tematyce, w naszym przykładzie – **ArkanoidKS** ●.

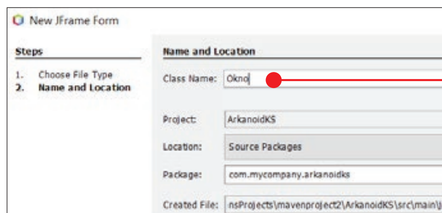
2 Inaczej niż w poprzednim rozdziale tym razem nie dodajemy do naszego projektu formatki, czyli **JFrame**, ale za to umieszczamy w naszym projekcie **JPanel Form** ●.

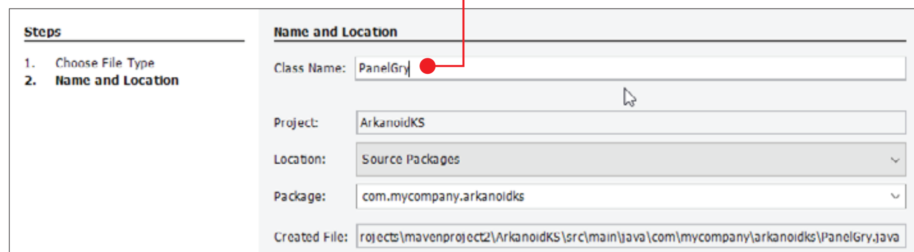


3 Nadajemy mu nazwę **PanelGry** ●. JPanel to płaszczyzna, na której będziemy mogli rysować. (Oczywiście poprzez „rysowanie" należy rozumieć używanie poleceń, które mogą umieszczać na płaszczyźnie kształty). Umieścimy na nim figury geometryczne, budując w ten sposób wygląd naszej gry. Dopiero tę płaszczyznę z figurami umieścimy na formatce. To oznacza, że również JFrame Form powinien się znaleźć w naszym projekcie. Dlatego klikamy prawym przyciskiem myszy na paczkę projektu, gdzie wcześniej dodaliśmy JPanel (czyli plik **PanelGry.java**) i tym razem dodajemy **JFrame Form** ●.



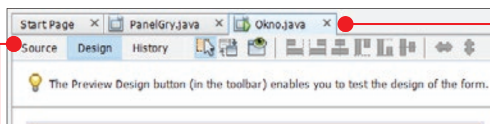
4 Nazywamy formatkę **Okno** ●. Skrypty naszej gry będziemy pisać zatem w dwóch





plikach: **PanelGry.java** – gdzie napiszemy, co ma się wyświetlać na panelu, i **Okno.java** – gdzie napiszemy, aby ten panel był integralną częścią okna naszej gry. W przypadku tego projektu nie musimy wykorzystywać kontrolek, które widzimy w palecie po prawej stronie okna po dodaniu formatki.

5 Dodane do projektu w poprzednich krokach pliki są automatycznie otwierane.



Możemy się między nimi przełączać, klikając na zakładki ● widoczne nad obszarem edycji. Zarówno w jednym, jak i drugim pliku musimy edytować skrypty – znajdziemy je, klikając na przycisk **Source** ● widoczny po przejściu do każdego z plików.

Umieszczanie panelu w oknie gry

1 Edycję zaczynamy od pliku **PanelGry.java**. Domyślnie, jak było widać na podglądzie, JPanel i JFrame mają kolor szary. Jeśli nasz panel zmieni kolor tła – to po dodaniu go do okna gry i uruchomieniu jej – będziemy widzieć, czy to, co napisaliśmy, zadziałało prawidłowo.

2 W wygenerowanym automatycznie skrypcie panelu znajdujemy jego konstruktor ●, czyli specjalną metodę klasy odpowiedzialną za tworzenie obiektów. Gdy jest ona wywoływana – powstaje obiekt. Zatem umieszczając coś w treści konstruktora, wiemy, że zostanie to wykonane, gdy będziemy tworzyć obiekt.

```
public PanelGry() {
    initComponents();

}
```

3 Dopisujemy do konstruktora polecenie **setBackground(Color.black);** ●. Zmienia ono kolor tła panelu na czarny.

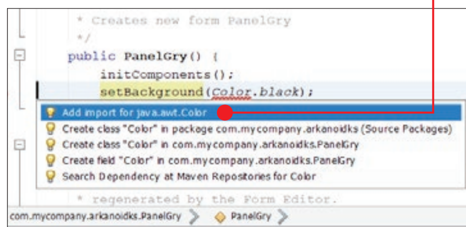
```
public PanelGry() {
    initComponents();
    setBackground(Color.black);
}
```

4 Po jego napisaniu słowo **Color** zostaje podkreślone na czerwono. W ten sposób wykorzystywane przez nas IDE zaznacza błędy w kodzie. Sam skrypt jest poprawny, jednak brakuje w nim części, która sprawi, że słowo **Color** będzie zrozumiałe dla programu. Chodzi o import odpowiedniej biblioteki. Tak jak w przykładzie z poprzedniego rozdziału, chcąc skorzystać z klas, które wymagają importu biblioteki, należy na początku skryptu wpisać słowo **import** oraz nazwę biblioteki. NetBeans IDE może pomóc w importowaniu odpowiednich bibliotek.

5 Przy linijce kodu zawierającej podkreślony fragment pojawia się ikona ● żarówki

```
    initComponents();
    setBackground(Color.bl
```

z czerwoną tarczą z wykrzyknikiem. Klikamy na nią, a program wyświetla możliwe rozwiązania wykrytego błędu. Pierwsze w tym przypadku to: **Add import for java.awt.Color** ●. Wybieramy tę opcję.



6 Import ● odpowiedniej biblioteki zostanie automatycznie dodany do skryptu, przed definicją klasy, a po linijce rozpoczynającej się od słowa **package** – mówiącej o przestrzeni nazw projektu.

```
package com.mycompany.arkanoidks;

import java.awt.Color;
```

7 Przechodzimy do pliku **Okno.java**. W nim też odszukujemy konstruktor ●.

```
public Okno() {
    initComponents();

}
```

8 Usuwamy z niego linijkę **initComponents();** ●, by nic spoza napisanego przez nas skryptu nie znalazło się w oknie gry.

```
public Okno() {
    initComponents();

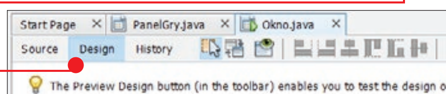
}
```

9 Dopisujemy do konstruktora polecenie **add(new PanelGry());** ●, które dodaje panel do okna.

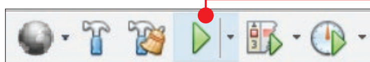
```
public Okno() {
    add(new PanelGry());
```

```
public Okno() {
    add(new PanelGry())
    pack();
```

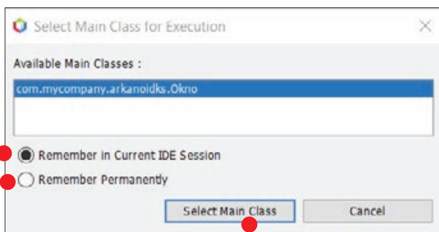
10 W kolejnej linijce wpisujemy polecenie **pack();** ●. Sprawi ono, że wielkość okna zostanie dopasowana do wielkości panelu. A wielkość panelu możemy zmieniać zarówno z poziomu skryptu, jak i z poziomu edytora graficznego pod opcją **Design** ●.



11 Klikając na **Run** ●, uruchamiamy napisany program, aby przetestować działanie stworzonego do tej pory skryptu.

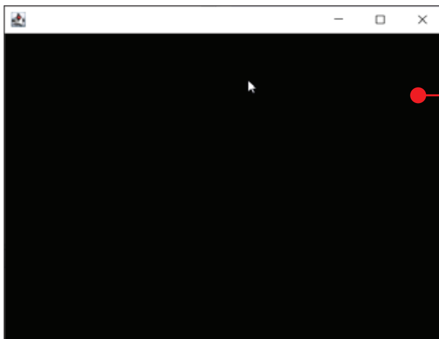


12 Przed uruchomieniem NetBeans IDE poprosi nas o wskazanie klasy głównej, czyli tej, której metoda **main** wykona się jako pierwsza po wystartowaniu aplikacji. W naszym wypadku będzie to klasa **Okno** – gdyż tylko w niej mamy metodę main. NetBeans pozwala nam jeszcze określić, czy klasę główną ustalamy tylko na bieżącą sesję IDE ● (do ponownego uruchomienia narzędzia),

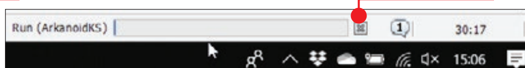


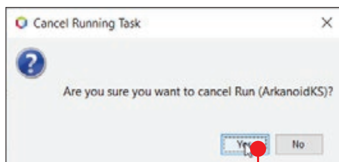
czy na stałe ●. Pozostańmy przy domyślnej opcji pierwszej. Ostateczne uruchomienie programu nastąpi po kliknięciu na przycisk **Select Main Class** ●.

13 Jeśli zobaczymy okno aplikacji wypełnione czarnym kolorem ●, to oznacza, że wszystko do tej pory zostało napisane poprawnie.



14 Po zamknięciu okna zwróćmy uwagę na pasek na dole. Mimo że okno tworzonego właśnie Arkanoida zostało zamknięte, sam program nadal działa w tle. Aby go wyłączyć, klikamy na znak **x** ● obok paska postępu widocznego u dołu okna NetBeans





IDE. Następnie zatwierdzamy wyłączenie go, klikając na przycisk **Yes** w nowym oknie. Dopiero wtedy program zostanie całkowicie wyłączony.

15 Tymczasem program powinien wyłączać się wraz z zamknięciem jego okna. By tak się działo, trzeba dodać jeszcze linijkę: **setDefaultCloseOperation(JFrame. EXIT ON CLOSE);**. Po tej zmianie, gdy uruchomimy program, a potem zamkniemy okno, zostanie on całkowicie wyłączony.

```
public Okno() {
    add(new PanelGry());
    pack();
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Dostosowywanie okna gry

W dalszej części realizacji projektu zobaczymy, jak z poziomu skryptu ustalić konkretną wielkość okna, określoną co do piksela.

```
public Okno() {
    add(new PanelGry());
    pack();
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setResizable(false);
}
```

1 Najpierw sprawimy, by użytkownik programu nie mógł ręcznie modyfikować ustalonej przez nas wielkości. W tym celu trzeba dopisać linijkę **setResizable(false);**. Po wprowadzeniu tej zmiany możemy zauważyć, że gdy program jest uruchomiony, przycisk do maksymalizacji okna jest nieaktywny. Nie da się też ręcznie zwęzić ani rozszerzyć okna.



2 Przechodzimy do **PanelGry.java**. To tu należy określić wymiary, ponieważ okno gry dostaje swoje wymiary zależnie od wymiarów panelu – wymiary panelu będą zatem wymiarami okna gry.

3 Wymiary okna będą nam potrzebne do dalszego tworzenia skryptów. Od nich zależy choćby to, gdzie na krawędzi okna ma się odbijać piłka. By to określić, będziemy musieli wykonać kilka działań matematycznych z wykorzystaniem właśnie wymiarów okna. Żeby mieć łatwy dostęp do tych wymiarów i móc je łatwo edytować, gdy zajdzie taka potrzeba, bez konieczności wprowadzania zmian w każdym miejscu kodu, w którym się do nich odnieślismy – umieścimy je w dwóch polach. Ponieważ nie przewidujemy zmieniania wymiarów okna gry podczas rozgrywki, możemy pola te utworzyć jako zmienne finalne, czyli takie, którym wartość w skrypcie nadawana jest tylko raz i nie jest później zmieniana. W jednym polu przechowamy szerokość, a w drugim wysokość panelu.
I tak, jeśli chcielibyśmy stworzyć zmienną do przechowania szerokości panelu, moglibyśmy napisać **int OKNO_SZER = 600;** A.

```
public class PanelGry extends javax.swing.JPanel {
        B        C
    private final int OKNO_SZER = 600; A
```

4 Jednak jeśli ma to być pole klasy, możemy określić jeszcze dostęp. Zgodnie z dobrą praktyką pole takie jest uznawane za wewnętrzne, żadna inna klasa nie musi wiedzieć, jak ono się nazywa, do czego służy i jaką ma wartość. Wystarczy, że wie to tworzący skrypt programista i odpowiednio wykorzysta pole

MODYFIKATORY DOSTĘPU

Dostęp określamy, stosując odpowiedni modyfikator. Wyróżniamy cztery modyfikatory dostępu. Najczęściej używane to **private** i **public**.

Pierwszy z nich oznacza, że pola czy metody są dostępne tylko w obrębie klasy, do której należą. Niemożliwe jest zatem odczytanie wartości takiego pola czy wywołanie takiej metody dla obiektu, który został utworzony w innej klasie.

Drugi z wymienionych natomiast daje taki dostęp.

Nie są to jedyne modyfikatory. Jest też na przykład modyfikator **protected** używany w bardziej złożonych relacjach pomiędzy obiektami. Ma on znaczenie w przypadku dziedziczenia. Poprzedzone nim elementy są udostępnione dla danej klasy i jej podklas. Elementy oznaczone modyfikatorem protected są też dostępne dla innych klas w tym samym pakiecie.

Brak modyfikatora także określa zakres dostępu – jest to tak zwany dostęp **package**, czyli w obrębie pakietu (projektu).

Po co określać dostęp

Określanie dostępu to inaczej enkapsulacja lub hermetyzacja. Jest to sposób na ukrycie szczegółów implementacji klasy. To jeden z fundamentów programowania obiektowego. Pozwala na pełną kontrolę nad zachowaniem i stanem danego obiektu.

Dobrą praktyką programistyczną jest stosowanie najbardziej restrykcyjnych modyfikatorów dostępu, co sprowadza się do używania modyfikatora **private** dla wszystkich pól i metod, które powinny być uznane za wewnętrzne. Pozostałe elementy, które stanowią interfejs komunikacji obiektu, oznaczamy słowem kluczowym **public**.

– do określenia wymiarów panelu. Dlatego powinniśmy użyć modyfikatora **private** **B** – piszemy zatem: **private int OKNO_SZER = 600;**

5 Skoro pole to nie będzie zmieniać wartości, powinniśmy użyć zmiennej finalnej. Takich zmiennych używamy właśnie w takich wypadkach. Pełna definicja pola, któremu nie będzie można zmienić wartości w innym miejscu skryptu, powinna mieć postać: **private final int OKNO_SZER = 600;** **C**. Dobrą praktyką jest, by nazwy zmiennych finalnych pisać wielkimi literami. Deklarację takiego pola umieszczamy wewnątrz klasy, ale poza jakąkolwiek jej metodą.

6 Podobne pole powinniśmy stworzyć dla wysokości panelu: **private final int OKNO_WYS = 500;** ●.

```
public class PanelGry extends javax.swing.JPanel {

    private final int OKNO_SZER = 600;
    private final int OKNO_WYS = 500;
```

7 W tym momencie mamy już pola zawierające docelowe wymiary okna gry, które ma mieć 600 pikseli szerokości i 500 pikseli wysokości. Teraz należałoby skorzystać z tych pól i przypisać ich wartości do wymiarów okna. Zrobimy to, pisząc w konstruktorze: **setPreferredSize(new Dimension(OKNO_SZER, OKNO_WYS));** ●.

```
public PanelGry() {
    initComponents();
    setBackground(Color.black);
    setPreferredSize(new Dimension(OKNO_SZER, OKNO_WYS));
}
```

8 Słowo **Dimension** nie będzie rozpoznane przez program i zostanie podkreślone jako błąd. By podkreślenie znikło, niezbędne jest dodanie importu odpowiedniej biblioteki. Możemy to zrobić ręcznie,

wpisując **import java.awt.Dimension;**, lub jak poprzednio (patrz strona 24) – korzystając z podpowiedzi programu. Wielkość okna powinna zostać odpowiednio ustawiona, gdy gra zostanie uruchomiona.

```
package com.mycompany.arkanoidks;

import java.awt.Color;
import java.awt.Dimension;
```

służy polecenie **setTitle()**. Ponieważ pasek tytułowy jest integralną częścią okna gry, a nie panelu, polecenia tego używamy w pliku **Okno.java**. Możemy skorzystać z niego bezpośrednio w konstruktorze, tak jak z innych poleceń używanych wcześniej. Tytuł okna wpisujemy w nawiasie polecenia, umieszczając go w cudzysłowie. Całość może mieć zatem na przykład taką formę: **setTitle(”ArkanoidKS”);**.

9 Możemy wprowadzić jeszcze jedną modyfikację okna gry. Na pasku tytułowym obecnie nic nie ma, a powino być na nim widać tytuł gry. Do ustawienia napisu na pasku tytułowym

```
public Okno() {
    add(new PanelGry());
    pack();
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setResizable(false);
    setTitle("ArkanoidKS");
}
```

Rysowanie na panelu

paintComponent() – ta metoda jest potrzebna do rysowania na panelu. Jest już ona w klasie **JPanel**. To co musimy zrobić, to zmienić jej treść. Nie będziemy zastępować całej treści metody, a jedynie dopisywać do niej kolejne instrukcje.

1 Wewnątrz klasy **PanelGry**, a pod definicją konstruktora wpisujemy **@Override**, co oznacza, że następna opisana metoda zostaje nadpisana. Metoda ta musi istnieć w klasie bazowej, po której dziedziczy nasza klasa. Tak jest właśnie w tym przykładzie.

```
    setBackground(Color.black);
    setPreferredSize(new Dimension(OKNO_SZER, OKNO_WYS));
}

@Override
public void paintComponent(Graphics g) {
```

2 W kolejnej linii piszemy **public void paintComponent(Graphics g) {**, co oznacza odniesienie do publicznej metody **paintComponent**, która za parametr bierze obiekt **g** klasy **Graphics**. W tym wypadku nie musimy tworzyć tego obiektu w naszej klasie i deklarować go jako kolejnego pola. Możemy za to wewnątrz metody **paintComponent** wywoływać jego metody – te z kolei pozwalają na rysowanie różnych kształtów, co jest naszym celem.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
```

3 Program nie rozumie jeszcze słowa **Graphics**, co możemy zmienić, importując odpowiednią bibliotekę. W tym wypadku można to zrobić, pisząc **import java.awt.Graphics;**.

4 By nie utracić dotychczasowej definicji metody, możemy rozpocząć jej nową definicję od zawartości starej definicji. Zrobimy to, wywołując metodę poprzez jej nazwę poprzedzoną słowem **super** – całość

powinna mieć następującą postać: **super.paintComponent (g) ;**.

```
@Override
public void paintComponent(Graphics g)
    super.paintComponent(g);

}
```

5 Ponieważ **paintComponent()** jest metodą publiczną, dobra praktyka wymaga, by nie opisywać bezpośrednio w niej rysowania wszystkich elementów okna gry, ale stworzyć w tym celu metodę prywatną. Zatem wewnątrz nadpisywanej metody wywołujemy nową metodę poleceniem **rysuj (g) ;**, a poniżej tworzymy definicję nowej metody **rysuj** poprzez polecenie **private void rysuj(Graphics g) {**.

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    rysuj(g);
}

private void rysuj(Graphics g) {

}
```

6 Chcąc narysować prostokąt, musimy dla obiektu **g** wywołać polecenie **fillRect** lub polecenie **drawRect**. Rysują one różne prostokąty, pierwsze z nich tworzy prostokąt wypełniony kolorem, drugie zaś – tylko kontur prostokąta. W jednym i drugim poleceniu prostokąt rysowany jest na podstawie liczb podanych w nawiasie. Należy podać cztery liczby. Pierwsza z nich to odległość prostokąta od lewej krawędzi panelu, a druga to odległość prostokąta od górnej krawędzi panelu, trzecia to szerokość figury, a czwarta – wysokość.
Zatem, by narysować całkowicie wypełniony kolorem prostokąt oddalony od prawej krawędzi o 100 pikseli, a od górnej krawędzi okna o 200 pikseli, o szerokości 300 pikseli i wysokości 250 pikseli, należy użyć polecenia **g.fillRect(100, 200, 300, 250);**.

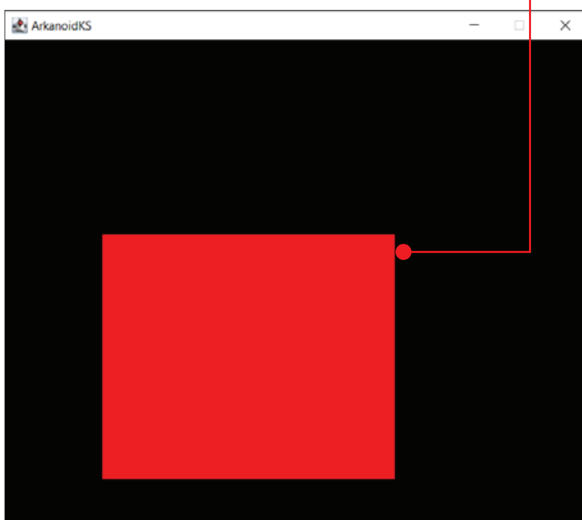
```
private void rysuj(Graphics g) {
    g.fillRect(100, 200, 300, 250);
}
```

7 By prostokąt ten miał inny niż domyślny, szary kolor, trzeba zastosować polecenie **g.setColor(Color.red);**. Polecenie to zmienia kolor rysowania na czerwony (można też zastosować inne kolory, podając ich angielskie nazwy).

```
private void rysuj(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(100, 200, 300, 250);
}
```

Co ważne, polecenia tego trzeba użyć przed poleceniem rysowania, można to sobie wytłumaczyć tak, że najpierw wybieramy farby, a potem nimi malujemy.

8 Gdy uruchomimy program po wprowadzeniu opisanych zmian, w oknie powinien pojawić się czerwony prostokąt.



Tworzenie paletki

W poprzedniej części rozdziału stworzyliśmy prostokąt. Teraz utrwalimy sobie to, czego się nauczyliśmy, i za pomocą poznanych poleceń stworzymy paletkę. Zarówno jej położenie w oknie gry, jak i wymiary powinny być określone nazwami, tak by później nie było kłopotu z ich odczytaniem. Niektóre z wartości będą się zmieniać, jak na przykład odległość paletki od lewej krawędzi (na tym polega ruch paletki), a inne będzie trzeba odczytać choćby po to, by sprawdzić, czy poruszająca się po oknie gry piłka trafiła w paletkę i powinna się od niej odbić.

1 Tworzenie paletki rozpoczniemy od deklaracji pól klasy, których wartości wykorzystamy do narysowania paletki, poprzez **private int szerPaletki = 100;**. Tego pola nie tworzymy jako finalnego, ponieważ rozważając różne warianty rozbudowy gry, możemy zechcieć zwiększyć poziom trudności rozgrywki poprzez zmniejszanie paletki. A wtedy pole **szerPaletki** będzie wymagało zmienienia jego wartości.

```
public class PanelGry extends javax.swing.JPanel {

    private final int OKNO_SZER = 600;
    private final int OKNO_WYS = 500;
    private int szerPaletki = 100;
```

2 Podobnie powinniśmy postąpić z polem do przechowywania wysokości paletki, które również tworzymy jako liczbę całkowitą poprzez **private int wysPaletki = 10;**. Nasza przykładowa paletka ma mieć 10 pikseli wysokości, jednak możemy dowolnie modyfikować jej wymiary, umieszczając w kodzie inne wartości dla tworzonych pól.

```
private final int OKNO_SZER = 600;
private final int OKNO_WYS = 500;
private int szerPaletki = 100;
private int wysPaletki = 10;
```

3 Następnie stworzymy pola **xp** i **yp**. Posłużą one do określenia współrzędnych **x** i **y** paletki. Współrzędne te inaczej określamy jako odległości odpowiednio od lewej krawędzi okna i od górnej krawędzi okna.

4 Polu **xp** należy nadać taką wartość, by narysowana już paletka znajdowała się idealnie na środku okna, czyli by jej odległość zarówno od lewej, jak i prawej krawędzi okna była taka sama. Ten efekt uzyskamy, gdy od szerokości okna (panelu) odejmiemy szerokość paletki i podzielimy wynik przez dwa. Dlatego deklarujemy pole linią **private int xp = (OKNO_SZER - szerPaletki) / 2;**. Nawet jeżeli wynik działania nie będzie liczbą całkowitą, część ułamkowa zostanie odcięta.

```
public class PanelGry extends javax.swing.JPanel {

    private final int OKNO_SZER = 600;
    private final int OKNO_WYS = 500;
    private int szerPaletki = 100;
    private int wysPaletki = 10;
    private int xp = (OKNO_SZER - szerPaletki) / 2;
```

5 Pole **yp** musi dostać taką wartość, by paletka swoją dolną krawędzią dotykała dolnej krawędzi okna. Zatem odległość paletki od górnej krawędzi okna można obliczyć, odejmując od wysokości okna (panelu) wysokość paletki. Deklaracji dokonujemy linią: **private int yp = OKNO_WYS - wysPaletki;**.

```
private int xp = (OKNO_SZER - szerPaletki) / 2;
private int yp = OKNO_WYS - wysPaletki;
```

6 Teraz możemy skorzystać z wartości tych pól do narysowania prostokąta będącego paletką. Mamy już utworzoną metodę **rysuj**. Dla zachowania przejrzystości kodu rysowanie poszczególnych elementów gry powinno być umieszczone w specjalnie dla nich stworzonych metodach, a metoda **rysuj** powinna być odpowiedzialna za zbiorowe

```
private void rysuj(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(100, 200, 300, 250);
}
private void rysujPaletke(Graphics g) {

}
```

wywołanie wszystkich pozostałych metod dotyczących rysowania. W takim wypadku należy stworzyć metodę **rysujPaletke()**, a jej wywołanie umieścić w metodzie **rysuj**, jednocześnie pozbywając się instrukcji odpowiedzialnych za rysowanie testowego prostokąta.

```
private void rysuj(Graphics g) {
    rysujPaletke(g);
}
private void rysujPaletke(Graphics g) {

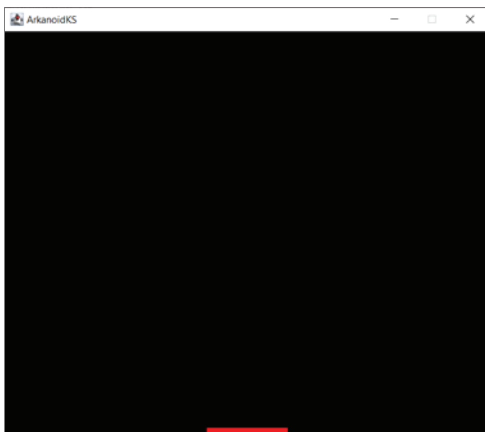
}
```

7 Treść metody **rysujPaletke** powinna rozpoczynać się od wyboru jej koloru. W naszym przykładzie paletka ma być czerwona z białą obwódką. Zaczniemy od koloru wypełnienia – poprzez polecenie **g.setColor(Color.red);** ustawiamy kolor rysowania na czerwony.

```
private void rysuj(Graphics g) {
    rysujPaletke(g);
}
private void rysujPaletke(Graphics g) {
    g.setColor(Color.red);
```

8 W kolejnej linii należy użyć polecenia **g.fillRect**, jednak inaczej niż w wypadku wcześniej rysowanego prostokąta w nawiasie po poleceniu nie wpisujemy liczb, lecz nazwy pól, z których wartości zostaną wykorzystane przez polecenie. Należy zatem wpisać: **g.fillRect(xp, yp, szerPaletki, wysPaletki);**.

```
private void rysujPaletke(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(xp, yp, szerPaletki, wysPaletki);
```



9 Jeśli teraz uruchomimy program, zobaczymy, że u dołu okna jest już widoczna paletka. Nie jest ona jednak jeszcze w pełni zgodna z założeniami – brakuje jej białej obwódki.

10 W kolejnej linii kodu należy zatem ustawić kolor rysowania na biały **g.setColor(Color.white);**.

```
private void rysujPaletke(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(xp, yp, szerPaletki, wysPaletki);
    g.setColor(Color.white);
```

11 Następnie, aby stworzyć obwódkę prostokąta, zamiast polecenia **fillRect** użyjemy polecenia **drawRect** (z tymi samymi parametrami). Zatem należy napisać: **g.drawRect(xp, yp, szerPaletki, wysPaletki);**.

```
private void rysujPaletke(Graphics g) {
    g.setColor(Color.red);
    g.fillRect(xp, yp, szerPaletki, wysPaletki);
    g.setColor(Color.white);
    g.drawRect(xp, yp, szerPaletki, wysPaletki);
}
```

12 Po uruchomieniu programu zobaczymy, że paletka ma białą obwódkę.



Tworzenie piłki

Sposób tworzenia piłki jest bardzo podobny do tworzenia paletki. Najpierw należy zadeklarować odpowiednie pola, a później skorzystać z nich w metodzie odpowiedzialnej za rysowanie piłki i zadbać o jej wywołanie, by rysunek faktycznie pojawił się w oknie gry.

```
private final int OKNO_SZER = 600;
private final int OKNO_WYS = 500;
private int szerPaletki = 100;
private int wysPaletki = 10;
private int xp = (OKNO_SZER - szerPaletki) / 2;
private int yp = OKNO_WYS - wysPaletki;
private int srednicaPilki = 20;
```

1 Piłka w przeciwieństwie do paletki jest okrągła, zatem do określenia jej wielkości wystarczy jeden wymiar, czyli jej średnica. Do przechowania tej wielkości możemy stworzyć pole **srednicaPilki** i nadać mu odpowiednią wartość, na przykład **20**.

2 Do narysowania piłki będą jeszcze potrzebne jej współrzędne **x** i **y**, czyli odległości od krawędzi okna.

```
private final int OKNO_SZER = 600;
private final int OKNO_WYS = 500;
private int szerPaletki = 100;
private int wysPaletki = 10;
private int xp = (OKNO_SZER - szerPaletki) / 2;
private int yp = OKNO_WYS - wysPaletki;
private int srednicaPilki = 20;
private int xPilki = (OKNO_SZER - srednicaPilki) / 2;
```

Odległość od lewej krawędzi okna można zapisać jako **xPilki**. Aby piłka znajdowała się idealnie na środku okna gry, trzeba dokonać podobnych obliczeń, jak w przypadku odległości paletki od lewej krawędzi okna, jednak w tym wypadku od szerokości panelu należy odjąć szerokość piłki, czyli jej średnicę, a wynik podzielić przez 2.
Zatem deklaracja współrzędnej **x** dla piłki powinna mieć postać: **private int xPilki = (OKNO_SZER - srednicaPilki) / 2;**.

3 Podobnie będzie z obliczeniem odległości piłki od górnej krawędzi okna, jednak tym razem zamiast szerokości okna do obliczeń należy wykorzystać jego wysokość. Linia kodu z deklaracją pola powinna zatem wyglądać tak: **private int yPilki = (OKNO_WYS - srednicaPilki) / 2;**.

```
private final int OKNO_SZER = 600;
private final int OKNO_WYS = 500;
private int szerPaletki = 100;
private int wysPaletki = 10;
private int xp = (OKNO_SZER - szerPaletki) / 2;
private int yp = OKNO_WYS - wysPaletki;
private int srednicaPilki = 20;
private int xPilki = (OKNO_SZER - srednicaPilki) / 2;
private int yPilki = (OKNO_WYS - srednicaPilki) / 2;
```

4 Mamy już zadeklarowane pola, ale żeby móc z nich skorzystać, należy stworzyć metodę przeznaczoną do rysowania piłki. Może się nazywać **rysujPilke**.

```
    private void rysujPilke(Graphics g) {

    }
```

5 Treść tej metody w naszym przykładzie będzie się zaczynać od wyboru koloru. Można przyjąć, że piłka tak jak paletka będzie czerwona z białą obwódką. I tu podobnie jak w wypadku paletki zaczynamy od wyboru koloru wypełnienia – **g.setColor(Color.red);**.

```
private void rysujPilke(Graphics g) {
    g.setColor(Color.red);
}
```

6 W kolejnej linii użyjemy polecenia **fillOval**, które działa bardzo podobnie do **fillRect**, tyle że wypełnia owal, a nie prostokąt. A owal o tej samej szerokości i wysokości da nam koło. Dlatego należy napisać **g.fillOval(xPilki, yPilki, srednicaPilki, srednicaPilki); A**.

```
private void rysujPilke(Graphics g) {
    g.setColor(Color.red);
A   g.fillOval(xPilki, yPilki, srednicaPilki, srednicaPilki);
}
```

```
private void rysujPilke(Graphics g) {
    g.setColor(Color.red);
    g.fillOval(xPilki, yPilki, srednicaPilki, srednicaPilki);
    g.setColor(Color.white);B
}
```

```
private void rysujPilke(Graphics g) {
    g.setColor(Color.red);
    g.fillOval(xPilki, yPilki, srednicaPilki, srednicaPilki);
    g.setColor(Color.white);
C   g.drawOval(xPilki, yPilki, srednicaPilki, srednicaPilki);
}
```

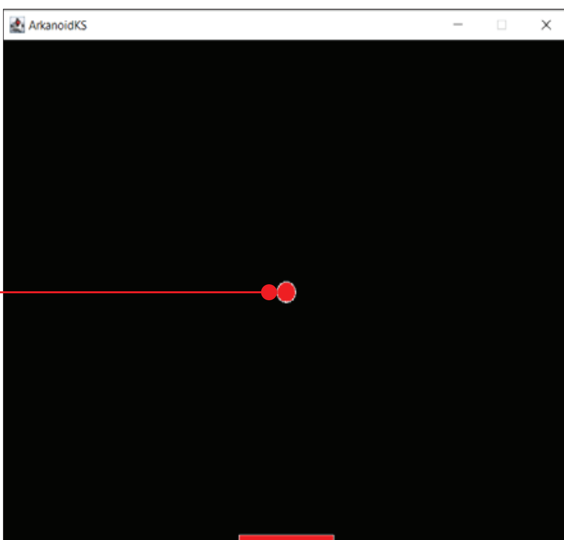
7 Następnie wybieramy biały jako kolor obwódki – **g.setColor(Color.white); B**.

8 Obwódkę tworzymy poleceniem **g.drawOval(xPilki, yPilki, srednicaPilki, srednicaPilki); C**.

9 Tak zdefiniowaną metodę należy wywołać w zbiorczej metodzie **rysuj**.

```
private void rysuj(Graphics g)
    rysujPaletke(g);
    rysujPilke(g);
}
```

10 Po uruchomieniu programu na środku okna gry zobaczymy piłkę.



Tworzenie cegieł

Mamy już narysowane paletkę i piłkę, ostatnim elementem graficznym, który jest nam potrzebny do stworzenia gry, są bloki (cegły), które będą w tej grze zbijane.

1 Żeby zająć się rysowaniem cegieł, niezbędne jest określenie ich liczby. Cegły powinny być ułożone regularnie, w określonej liczbie kolumn i wierszy. Aby gra była bardziej nieprzewidywalna, liczba kolumn i wierszy może przyjąć wartość pseudolosową. A żeby taką wartość uzyskać, niezbędne jest stworzenie obiektu klasy **Random** pełniącego funkcję generatora liczb pseudoloso-

wych. Należy zatem napisać: **private Random generator = new Random();**.

```
private final int OKNO_SZER = 600;
private final int OKNO_WYS = 500;
private int szerPaletki = 100;
private int wysPaletki = 10;
private int xp = (OKNO_SZER - szerPaletki) / 2;
private int yp = OKNO_WYS - wysPaletki;
private int srednicaPilki = 20;
private int xPilki = (OKNO_SZER - srednicaPilki) / 2;
private int yPilki = (OKNO_WYS - srednicaPilki) / 2;
private Random generator = new Random();
```

2 By klasa **Random** była rozpoznana przez program, trzeba dokonać importu biblioteki: **import java.util.Random;**.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.util.Random;
```

3 Aby skorzystać z tego generatora, należy utworzyć pole reprezentujące liczbę kolumn, w jakich mają być ułożone cegły, a także pole reprezentujące liczbę ich wierszy. Wartość tych pól powinna być nadana poprzez polecenie **nextInt**. Przyjmijmy, że liczba wierszy i kolumn powinna mieścić się w zakresie 4–8. Polecenie **nextInt** pozwala jedynie na określenie górnej granicy generowania liczby. Można sobie z tym poradzić – wystarczy wygenerować liczbę w zakresie do 5 (5 jest wartością graniczną – ale nie może zostać wygenerowana, generowana jest „liczba mniejsza od 5") i do tej wygenerowanej liczby dodać 4. Wtedy otrzymamy liczbę z oczekiwanego przez nas zakresu. Definicja pola do przechowywania liczby kolumn powinna mieć postać **private int kolumny = generator.nextInt(5) + 4;**, a do liczby wierszy **private int wiersze = generator.nextInt(5) + 4;**.

```
private int yPilki = (OKNO_WYS - srednicaPilki) / 2;
private Random generator = new Random();
private int wiersze = generator.nextInt(5) + 4;
private int kolumny = generator.nextInt(5) + 4;
```

4 Każda cegła powinna być taka sama, więc wszystkie mają mieć identyczną szerokość i wysokość. By tak się stało, ich wymiary powinny również trafić do pól klasy. Należy zatem stworzyć linię kodu: **private int szerCegla = 50;**. Określając szerokość cegły, warto mieć na uwadze to, jaka jest maksymalna liczba kolumn i jaka jest szerokość okna. Szerokość cegły pomnożona

```
private int yPilki = (OKNO_WYS - srednicaPilki) / 2;
private Random generator = new Random();
private int wiersze = generator.nextInt(5) + 4;
private int kolumny = generator.nextInt(5) + 4;
private int szerCegla = 50;
```

przez liczbę kolumn (maksymalną) – w tym wypadku 8*50 = 400 – powinna być mniejsza niż szerokość okna – w tym wypadku 500. Różnica pomiędzy tymi wartościami musi być na tyle duża, by zmieściły się też odstępy pomiędzy cegłami. Dlatego po zadeklarowaniu wysokości cegieł (linia: **private int wysCegla = 10;**) należy zająć się deklaracją odstępów pomiędzy tymi cegłami.

```
private int szerCegla = 50;
private int wysCegla = 10;
```

5 Możemy wyróżnić przynajmniej dwa rodzaje odstępów: jeden to odległość pomiędzy kolumnami, a drugi to odległość między wierszami. A warto także pamiętać o odległości pierwszej kolumny cegieł od lewej krawędzi okna i odległości pierwszego wiersza cegieł od górnej krawędzi okna. Zacznijmy od odległości między cegłami. Możemy uznać, że zarówno między kolumnami, jak i między wierszami może być taka sama odległość.
Aby ją określić, deklarujemy pole **private int ceglyOdstep = 5;** **A**.

```
private int kolumny = generator.nextInt(5) + 4;
private int szerCegla = 50;
private int wysCegla = 10;
private int ceglyOdstep = 5;
```
A

6 Następnie można określić odstęp pierwszej kolumny od lewej krawędzi okna. Ta odległość powinna być taka sama jak odległość ostatniej kolumny od prawej krawędzi okna. By obliczyć ten odstęp, należy policzyć całą szerokość dużego prostokąta, który budują wszystkie cegły. Będzie ona równa sumie szerokości wszystkich kolumn, do której powinny zostać dodane wszystkie odstępy między kolumnami. A odstępów tych jest zawsze o jeden mniej niż kolumn. Taką wspólną szerokość cegieł należy odjąć od szerokości okna – a wynik podzielić przez dwa. Pole powstałe w wyniku polecenia **private int ceglyOdstepLewa = (OKNO_SZER - (szerCegla*kolumny + ceglyOdstep*(kolumny-1))) / 2;** będzie zawierać taką odległość od krawędzi, jakiej poszukujemy.

```
private Random generator = new Random();
private int wiersze = generator.nextInt(5) + 4;
private int kolumny = generator.nextInt(5) + 4;
private int szerCegla = 50;
private int wysCegla = 10;
private int ceglyOdstep = 5;
private int ceglyOdstepLewa = (OKNO_SZER - (szerCegla*kolumny + ceglyOdstep*(kolumny-1))) / 2;
```

```
private int ceglyOdstep = 5;
private int ceglyOdstepLewa = (OKNO_SZER - (szerCegla*kolumny + ceglyOdstep*(kolumny-1))) / 2;
private int ceglaOdstepGora = 10;
```

7 Ostatni z odstępów to odległość pomiędzy pierwszym wierszem cegieł a górną krawędzią okna. Ta wielkość nie musi wynikać z obliczeń i może być ustalona przez programistę, na przykład w formie: **private int ceglaOdstepGora = 10;**.

8 Kolejną ważną rzeczą jest lokalizacja cegieł. Każda cegła będzie miała inną lokalizację. Nie oznacza to jednak, że dla każdej cegły należy tworzyć oddzielne pole do przechowywania jej współrzędnej **x** i oddzielne pole dla jej współrzędnej **y**. Po dwa pola dla każdej z maksymalnie 64 cegieł – to razem byłoby 128 pól. To zdecydowanie nie byłoby dobre rozwiązanie. Na szczęście z pomocą przychodzą nam tablice. Przypominają one zmienne, jednak pozwalają przechowywać pod jedną nazwą więcej różnych ponumerowanych wartości. Tablice mogą być wielowymiarowe, na przykład dwuwymiarowe. Wtedy każda wartość w takiej tablicy ukrywa się nie pod jednym numerem, ale pod dwoma. I takie rozwiązanie będzie dla nas przydatne. Skorzystamy z tablic dwuwymiarowych do przechowywania współrzędnych cegieł. Każda cegła będzie identyfikowana numerem kolumny i wiersza, w których się znajduje. Stworzenie tablicy o nazwie **ceglyX** do przechowywania współrzędnych **x** wszystkich cegieł wymaga napisania linii: **private int ceglyX[][] = new int[kolumny][wiersze];** B. Liczba kwadratowych nawiasów po nazwie tablicy mówi o liczbie jej wymiarów. Wartości wpisane w kwadratowe nawiasy po typie danych mówią o maksymalnej liczbie elementów tablicy. A liczba ta wynika z liczby kolumn i wierszy. Opisana w tym punkcie linia kodu tworzy jedynie tablice – konkretne wartości zostaną w tej tablicy umieszczone w dalszych krokach.

9 Analogicznie do tablicy służącej do przechowywania współrzędnych **x** wszystkich cegieł należałoby stworzyć odpowiednią

```
private int ceglyOdstepLewa = (OKNO_SZER - (szerCegla*kolumny + ceglyOdstep*(kolumny-1))) / 2;
private int ceglaOdstepGora = 10;
private int ceglyX[][] = new int[kolumny][wiersze];
```
B

```
public PanelGry() {
    initComponents();
    setBackground(Color.black);
    setPreferredSize(new Dimension(OKNO_SZER, OKNO_WYS));

  A for (int i = 0; i < kolumny; i++) {
      B for (int j = 0; j < wiersze; j++) {
          C ceglyX[i][j] = i * (szerCegla + ceglyOdstep) + ceglyOdstepLewa;
          D ceglyY[i][j] = j * (wysCegla + ceglyOdstep) + ceglaOdstepGora;
      E }
  F }
}
```

tablicę do przechowywania współrzędnych **y** wszystkich cegieł. Można to zrobić w ten sposób: **private int ceglyY[][] = new int[kolumny][wiersze];**.

```
private int ceglyX[][] = new int[kolumny][wiersze];
private int ceglyY[][] = new int[kolumny][wiersze];
```

10 Nadając wartości poszczególnym elementom tablic stworzonych w poprzednich krokach, trzeba odnosić się do każdego z nich z osobna. Czy będzie to wymagało napisania 128 linii kodu, po jednej dla każdej wartości? Na szczęście nie, a to dzięki temu, że mamy do dyspozycji pętle, czyli konstrukcje pozwalające wielokrotnie wykonywać instrukcje. W tym wypadku pomocna okaże się pętla **for**. Pozwala ona na wykonanie wybranych instrukcji określoną liczbę razy i ma następującą składnię: **for (int i = 0; i < kolumny; i++) { zapętlone instrukcje }**, gdzie:

- **int i = 0** – oznacza deklarację zmiennej **i**, która iterując, zmienia swoją wartość; można w tej pętli korzystać z wartości zmiennej tworzonej na potrzeby pętli;
- **i < kolumny** – oznacza warunek wykonywania pętli, w tym wypadku zmienna **i** musi mieć wartość mniejszą niż zmienna **kolumny**. W przeciwnym wypadku wykonywanie instrukcji z pętli zostanie zakończone;
- **i++** – oznacza sposób zmiany wartości zmiennej **i** z każdym przejściem pętli; zapis ten jest tożsamy z zapisem **i = i+1**.

Nadanie wartości elementom tablicy należy wpisać w konstruktorze klasy **PanelGry**.

Powinno ono wyglądać następująco:

A – pętla **for**, która wykona się dla każdej kolumny;

B – pętla **for**, która wykona się dla każdego wiersza w kolumnie; pętla w pętli – w ten sposób poprzez zmienne **i** i **j** możemy odnieść się do każdej cegły i opisać jej lokalizację;

C – przypisanie wartości współrzędnej **x** dla elementu tablicy określonego numerami **[i][j]**. Działanie **i * (szerCegla + ceglyOdstep) + ceglyOdstepLewa** to wzór na obliczenie odległości poszczególnych cegieł od lewego brzegu okna. Dla pierwszej kolumny odległość jest taka sama jak wartość **ceglyOdstepLewa**, dlatego że **i** w pierwszym przejściu pętli jest równe zeru; dla każdej kolejnej kolumny odległość ta jest większa o szerokość cegły oraz odstęp po tej kolumnie;

D – przypisanie wartości współrzędnej **y** dla elementu tablicy określonego numerami **[i][j]**. Działanie **j * (wysCegla + ceglyOdstep) + ceglaOdstepGora** to wzór na obliczenie odległości poszczególnych cegieł od górnej krawędzi okna; dla pierwszego wiersza odległość jest taka sama jak wartość **ceglaOdstepGora**, dlatego że **j** w pierwszym przejściu pętli jest równe zeru. Dla każdego kolejnego wiersza odległość ta jest większa o wysokość cegły oraz odstęp po kolejnym wierszu;

E – zamknięcie wewnętrznej pętli **for**;

F – zamknięcie treści zewnętrznej pętli **for**.

11 Gdy współrzędne cegieł są już obliczone i zapisane w tablicach, można przejść do rysowania cegieł. W tym celu należy zdefiniować nową metodę – **rysujCegly**.

```
private void rysujCegly(Graphics g) {

}
```

12 Obliczając współrzędne każdej z cegieł, by się do niej odnieść, stosujemy dwie pętle **for** – jedna w drugiej – w taki sposób, by zmienne tworzone na potrzeby pętli odpowiadały numerom cegieł w dwóch wymiarach tablicy. Rysując cegły, znów należy się odnieść do każdej z nich. To oznacza ponowne zastosowanie pętli **for**. Pierwsza z nich – **for (int i = 0; i < kolumny; i++) { }** wykona się tyle razy, ile jest kolumn.

```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {

    }
}
```

13 Druga z pętli powinna znajdować się wewnątrz pierwszej i wykonać się tyle razy, ile mamy wierszy – **for (int j = 0; j < wiersze; j++) { }**.

```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
        }
    }
}
```

14 Wewnątrz wewnętrznej pętli możliwe jest już odnoszenie się do kolejnych elementów dwuwymiarowych tablic.

Należy je wykorzystać w poleceniu rysującym prostokąt. A zanim prostokąt zostanie narysowany, trzeba wybrać jego kolor. W naszym przykładzie cegły będą czerwonymi prostokątami z białymi obwódkami. Podobnie jak w przypadku poprzednich metod odpowiedzialnych za rysowanie zaczniemy od wyboru koloru wypełnienia – **g.setColor(Color.red);**.

```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
            g.setColor(Color.red);
        }
    }
```

15 Kolejna linia kodu to rysowanie prostokąta. W poleceniu **fillRect** należy użyć wartości współrzędnych przechowywanych w tablicach.
Cała linia kodu powinna mieć formę: **g.fillRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);**.

16 Kolejny krok to wybór białego jako koloru obwódki – **g.setColor(Color.white);**.

```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
            g.setColor(Color.red);
            g.fillRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);
        }
    }
}
```

```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
            g.setColor(Color.red);
            g.fillRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla).
            g.setColor(Color.white);
        }
    }
}
```

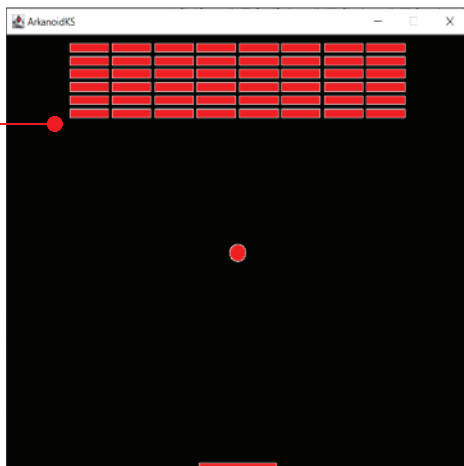
```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
                g.setColor(Color.red);
                g.fillRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);
                g.setColor(Color.white);
                g.drawRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);
        }
    }
}
```

17 Następnie należy narysować obwódkę poprzez polecenie **drawRect** w formie: **g.drawRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);** ●.

18 Stworzoną w ten sposób metodę należy wywołać ● w zbiorczej metodzie rysującej.

```
private void rysuj(Graphics g) {
    rysujPaletke(g);
    rysujPilke(g);
    rysujCegly(g);
}
```

19 Po uruchomieniu programu cegły (w wylosowanej liczbie kolumn i wierszy) są już widoczne ●.



Ukrywanie zbitych cegieł

Metoda **rysujCegly** podczas rysowania nie uwzględnia tego, czy rysowana cegła została już zbita, czy jeszcze nie. Należy stworzyć mechanizm, który pozwala na oznaczanie, czy cegła została trafiona i czy powinna być narysowana.

1 Informacja o tym, czy dana cegła została zbita, powinna znaleźć się w specjalnie do tego celu utworzonej tablicy dwuwymiarowej, której elementy będą numerowane tak jak w przypadku dwóch poprzednich tablic do przechowywania współrzędnych cegieł. Te same numery będą odnosiły się do tej samej cegły we wszystkich tablicach, z jakich skorzystamy w tym zastosowaniu.

Nowa tablica przechowuje inny rodzaj danych. Wcześniejsze tablice przechowywały liczby całkowite. A ta tablica powinna przechowywać zero-jedynkową informację o tym, czy dana cegła została już trafiona, czy nie. Nowa tablica powinna mieć więc typ danych **boolean**. Można ją stworzyć, pisząc linię kodu: **private boolean czyZbite[][] = new boolean[kolumny][wiersze];** ●.

```
private int ceglaOdstepGora = 10;
private int ceglyX[][] = new int[kolumny][wiersze];
private int ceglyY[][] = new int[kolumny][wiersze];
private boolean czyZbite[][] = new boolean[kolumny][wiersze];
```

```
public PanelGry() {
    initComponents();
    setBackground(Color.black);
    setPreferredSize(new Dimension(OKNO_SZER, OKNO_WYS));

    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
            ceglyX[i][j] = i * (szerCegla + ceglyOdstep) + ceglyOdstepLewa;
            ceglyY[i][j] = j * (wysCegla + ceglyOdstep) + ceglaOdstepGora;
            czyZbite[i][j] = false;
        }
    }
}
```

2 Podobnie jak w przypadku poprzednich tablic wartości poszczególnych elementów powinny zostać nadane w konstruktorze. W przypadku tej tablicy także należy się odnosić do każdej cegły z osobna. To oznacza konieczność wykorzystania dwóch pętli **for**. Wcale nie trzeba robić nowych pętli – można wykorzystać te stworzone wcześniej i w nich, oprócz obliczania współrzędnych, oznaczać cegły jako jeszcze nie zbite, bo taki powinien być ich stan początkowy.
Typ danych **boolean** pozwala na przechowywanie dwóch wartości: **true**, co oznacza prawdę, i **false**, co oznacza nieprawdę (fałsz).
Jeśli tablica nazywa się **czyZbite**, odpowiedzią na to pytanie byłoby false, ponieważ w stanie początkowym wszystkie cegły nie są jeszcze zbite.

3 Aby oznaczyć cegły jako jeszcze nie zbite, należy do wnętrza pętli dopisać: **czyZbite[i][j] = false;**.

4 By w oknie gry pojawiały się tylko te cegły, które nie są jeszcze zbite, trzeba zmodyfikować metodę **rysujCegly**. Wykonywanie ciągu instrukcji odpowiedzialnych za rysowanie każdej z cegieł powinno być uzależnione od wartości znajdującej się w tablicy **czyZbite** dla elementu odpowiadającego cegle, wynikającej z iterujących pętli.
Wykonywanie pewnych instrukcji od zaistniałych warunków możliwe jest przy wykorzystaniu instrukcji warunkowej – **if**. Zatem cztery linie kodu odpowiedzialne za rysowanie należy zamknąć w nawias klamrowy, trzeba je też poprzedzić słowem **if** wraz z nawiasem, w którym będzie zapisany warunek mówiący o tym, że cegła, do której odnosi się aktualnie sformułowanie **czyZbite[i][j]**, faktycznie nie była zbita. Robimy to, pisząc: **if (!czyZbite[i][j])**.

5 Dodanie tej instrukcji po uruchomieniu programu nie wprowadzi widocznych zmian. Jak w takim razie sprawdzić,

```
private void rysujCegly(Graphics g) {
    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
            if (!czyZbite[i][j]) {
                g.setColor(Color.red);
                g.fillRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);
                g.setColor(Color.white);
                g.drawRect(ceglyX[i][j], ceglyY[i][j], szerCegla, wysCegla);
            }
        }
    }
}
```

czy wszystko zostało napisane poprawnie? W konstruktorze, gdzie nadawana jest wartość początkowa dla elementów tablicy **czyZbite**, można zmienić false na **true** ●. Wtedy po ponownym uruchomieniu gry cegły staną się niewidoczne, co oznacza, że skrypt został napisany prawidłowo. Po wykonaniu tego testu należy przywrócić wartość **false** w tablicy.

```
for (int i = 0; i < kolumny; i++) {
    for (int j = 0; j < wiersze; j++) {
        ceglyX[i][j] = i * (szerCegla + c
        ceglyY[i][j] = j * (wysCegla + ceg
        czyZbite[i][j] = true;
    }
```

Ruch piłki

Elementy gry zostały już narysowane. Kolejnym krokiem jej realizacji powinno być wprowadzenie interakcji pomiędzy tymi elementami. W interakcje z cegłami i paletką wchodzi piłka. Bez jej ruchu w grze nic by się nie działo.

```
private int ceglyX[][] = new int[kolumny][wiersze];
private int ceglyY[][] = new int[kolumny][wiersze];
private boolean czyZbite[][] = new boolean[kolumny][wiersze];
private int dx = 2;
```

1 Zastanówmy się: w gruncie rzeczy „ruch" piłki jest zmianą jej położenia. A położenie może zmieniać się na osi X (w poziomie) i na osi Y (w pionie).
Ruch piłki musi być stały. Nie zatrzymuje się ona, a jedynie zmienia kierunek lotu, odbijając się od ścian, paletki i cegieł.
Dlatego do skryptu należy wprowadzić dwa nowe pola – **dx** i **dy**, które będą odpowiadały za „krok", jaki wykonuje piłka, odpowiednio na osi X i osi Y.
Inaczej mówiąc, z każdym kolejnym wywołaniem metody odpowiedzialnej za ruch piłki wartości tych pól zostaną dodane odpowiednio do współrzędnej X i współrzędnej Y piłki.
Gdy **dx** będzie miało wartość dodatnią, to na osi X piłka będzie się poruszać w prawą stronę, bo jej odległość od lewej krawędzi będzie rosnąć.
By piłka poruszała się w stronę lewą na osi X, trzeba nadać polu **dx** wartość ujemną. Natomiast w przypadku osi Y – gdy **dy** będzie miało wartość dodatnią, to piłka będzie poruszała się w dół, a gdy ujemną – w górę.

2 Pola **dx** i **dy** powinny być zadeklarowane jako liczby całkowite, a ich początkowe wartości powinny być takie, by na starcie gry piłka leciała w górę, a nie w dół. Można zapisać zatem **private int dx = 2;** ●, by piłka leciała w prawą stronę, i **private int dy = -2;** ●, by równocześnie przemieszczała się w górę.

```
private int dx = 2;
private int dy = -2;
```

3 Następnie trzeba zdefiniować nową metodę **ruchPilki** ●, a w jej treści dodawać wartości pól **dx** i **dy** do współrzędnych piłki.

```
private void ruchPilki() {

}
```

4 W treści metody **ruchPilki** należy wykonywać dodawanie wartości pól **dx** i **dy** odpowiednio do **xPilki** i **yPilki**. Zapisujemy to w taki sposób, by dla **xPilki** nowa wartość wynosiła dotychczasową wartość plus **dx**: **xPilki = xPilki + dx;** ●.

```
private void ruchPilki() {
    xPilki = xPilki + dx;
}
```

```
public class PanelGry extends javax.swing.JPanel implements ActionListener

    private final int OKNO_SZER = 600;
    private final int OKNO_WYS = 500;
    private int szerPaletki = 100;
    private int wysPaletki = 10;
```

5 Ze współrzędną **y** postępujemy analogicznie: **yPilki = yPilki + dy;**.

```
private void ruchPilki() {
    xPilki = xPilki + dx;
    yPilki = yPilki + dy;
}
```

6 Teraz zastanówmy się, kiedy należy wywoływać nową metodę. Powinno to mieć miejsce cyklicznie z pewnym odstępem czasu. By było to możliwe, nasza klasa będzie potrzebowała skorzystania z Timera. Obiekt tej klasy jest w stanie cyklicznie wywoływać działanie. Należy więc zadeklarować obiekt **timer** klasy **Timer** poleceniem: **private Timer timer;**.

```
private boolean czyZbi
private int dx = 2;
private int dy = -2;
private Timer timer;
```

7 By klasa **Timer** była rozpoznawana przez ten projekt, należy dokonać importu biblioteki: **import javax.swing.Timer;**.

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.util.Random;
import javax.swing.Timer;
```

8 Oprócz deklaracji obiektu trzeba jeszcze wywołać jego konstruktor, aby wytworzyć obiekt. Możemy to napisać w konstruktorze klasy **PanelGry**. Tam należałoby użyć linii: **timer = new Timer(10, this);**, gdzie liczba **10** odnosi się do interwału kolejnych wywołań (czas wyrażony w milisekundach), a słowo **this** – do bieżącej klasy.

```
                czyZbito[i][j] = false
            }
        }
        timer = new Timer(10, this);
```

9 Program podkreśla słowo **this** jako błąd. By zostało ono rozpoznane przez program w tym kontekście, do definicji klasy należy dodać implementację interfejsu **ActionListener**. Należy zatem nieco zmienić linię tworzącą klasę, by miała ona postać: **public class PanelGry extends javax.swing.JPanel implements ActionListener {**.

10 By interfejs ten był rozpoznany przez program, niezbędne jest dodanie importu – **import java.awt.event.ActionListener;**.

```
import java.awt.Graphics;
import java.util.Random;
import javax.swing.Timer;
import java.awt.event.ActionListener;
```

11 Tu program również zgłasza błąd – klasa, implementując interfejs, musi mieć w sobie wszystkie metody zawarte w danym interfejsie. To wymusza implementację metody **actionPerformed**. W jej wygenerowaniu może pomóc NetBeans. Klikamy na ikonę błędu obok definicji klasy. Z rozwiniętej listy możliwych rozwiązań trzeba wybrać opcję **Implement all abstract methods.**, co wygeneruje w klasie **PanelGry** wszystkie

```
20   */
     public class PanelGry extends javax.swing.JPanel implements ActionListener {
22     Implement all abstract methods
23     Make class PanelGry abstract         _SZER = 600;
24       private final int OKNO_WYS = 500;
         private int szerPaletki = 100;
```

```
@Override
public void actionPerformed(ActionEvent e) {
    throw new UnsupportedOperationException("Not supported yet."); //T
}
```

metody wymagane implementowanym interfejsem. W przypadku tego interfejsu jest to jedna metoda – **actionPerformed**.

12 W parametrze tej metody mamy obiekt klasy **ActionEvent**, która nie jest rozpoznawana przez ten projekt. By została rozpoznana, należy dokonać importu odpowiedniej biblioteki – **import java.awt.event.ActionEvent;**.

```
import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

13 To, co znajduje się w definicji metody **actionPerformed**, będzie przez timer wykonywane z określonym wcześniej interwałem. Należy zmienić treść wygenerowanej metody, usuwając z jej treści znajdującą się tam linię kodu.

```
@Override
public void actionPerformed(ActionEvent e) {

}
```

14 W treści metody **actionPerformed** powinna być wywoływana metoda **ruchPilki**.

```
@Override
public void actionPerformed(ActionEvent e) {
    ruchPilki();
}
```

15 Uruchomienie programu na tym etapie jeszcze nie da efektu poruszającej się piłki. To dlatego, że program jedynie zmienia logiczne wartości lokalizacji piłki, a piłka nadal jest narysowana w swojej lokalizacji początkowej. Konieczne zatem jest ponowne narysowanie elementów okna. Można to osiągnąć za pomocą polecenia **repaint**, które (w skrócie) prowadzi do wywołania metody **paintComponent**, czyli tej, w której wywołujemy rysowanie elementów gry.

```
@Override
public void actionPerformed(ActionEvent e) {
    ruchPilki();
    repaint();
}
```

16 Na razie jednak piłka jeszcze się nie poruszy, a to dlatego, że timer nie został uruchomiony. Żeby go uruchomić, trzeba użyć polecenia **timer.start();**. Może ono zostać wpisane w konstruktor. Po tej operacji, po uruchomieniu programu, piłka zacznie przemieszczać się w kierunku wynikającym z wartości pól **dx** i **dy**.

```
    timer = new Timer(10, this);
    timer.start();
```

Odbijanie piłki

Metoda **ruchPilki** działa teraz tak, że cały czas przesuwa piłkę w kierunku określonym przez pole **dx** i **dy**, natomiast nie bierze pod uwagę jej lokalizacji – kierunek nie zmienia się, gdy piłka dotrze do krawędzi okna. Po prostu wylatuje ona z okna gry. Należy tak zmodyfikować definicję metody, by w takim wypadku kierunek ruchu się zmieniał.

```
private void ruchPilki() {
    if (xPilki<=0 || xPilki > OKNO_SZER - srednicaPilki){
        dx = dx*-1;
    }
    if (yPilki <= 0){

    }
    xPilki = xPilki + dx;
    yPilki = yPilki + dy;
}
```

1 Przed zmianą kierunku piłki należy sprawdzić, czy dotyka ona krawędzi okna. Takiego sprawdzenia można dokonać poprzez instrukcję warunkową. Dla lewej krawędzi okna, gdy **xPilki** (czyli odległość piłki od lewej krawędzi) będzie mniejsze lub równe zeru – **dx** powinno zostać pomnożone przez -1, co jest równoznaczne ze zmianą kierunku piłki.

```
private void ruchPilki() {
    if (xPilki<=0){
        dx = dx*-1;
    }
    xPilki = xPilki + dx;
    yPilki = yPilki + dy;
}
```

2 Instrukcję warunkową możemy zmodyfikować tak, by dodać do niej drugi warunek. Jeśli warunki będą połączone spójnikiem „lub”, do wykonania zmiany kierunku wystarczy, że jeden z warunków zostanie spełniony. Drugi warunek powinien dotyczyć dotarcia przez piłkę do prawej krawędzi. Wydawać by się mogło, że piłka dotyka prawej krawędzi okna, gdy jej odległość od lewej krawędzi jest równa szerokości okna. Jednak wtedy piłka jest już poza oknem. Piłka swoją prawą krawędzią dotyka prawej krawędzi okna, gdy odległość piłki od lewej krawędzi zrówna się z szerokością okna pomniejszoną o szerokość piłki (czyli jej średnicę). Warunek powinien mieć zatem postać: **xPilki > OKNO_SZER - srednicaPilki**. Do połączenia warunków spójnikiem „lub”, używamy dwóch pionowych kresek – **||**.

```
private void ruchPilki() {
    if (xPilki<=0 || xPilki > OKNO_SZER - srednicaPilki){
        dx = dx*-1;
    }
    xPilki = xPilki + dx;
    yPilki = yPilki + dy;
}
```

3 Podobne odbijanie należy opisać także dla osi Y – piłka ma odbijać się też od górnej krawędzi okna i od paletki. Poprzez instrukcję warunkową trzeba sprawdzić, czy odległość piłki od górnej krawędzi jest równa zeru (bądź mniejsza od zera) . Wtedy piłka powinna zmienić swój kierunek ruchu na osi Y. Piszemy zatem **if (yPilki <= 0){ }**, by utworzyć warunek, i **dy = dy*-1;**, by zmienić kierunek ruchu piłki.

```
if (yPilki <= 0){
    dy = dy*-1;
}
```

4 Podobnie jak poprzednio do odbijania piłki na osi Y też możemy użyć jednej instrukcji warunkowej – z połączonymi spójnikiem warunkami. Jednak w przypadku dolnej krawędzi okna warunki muszą być nieco inne:

- Piłka musi mieścić się w zasięgu paletki na osi X. Zatem jej odległość od lewej krawędzi okna musi być większa niż odległość paletki od lewej krawędzi okna. Jednocześnie odległość piłki od lewej krawędzi okna nie może być większa niż odległość prawej krawędzi paletki od lewej krawędzi okna – wszystko to możemy zapisać jako dwa warunki: **xPilki > xp** i **xPilki < xp + szerPaletki** – co istotne, muszą być one spełnione jednocześnie – takie warunki łączy się spójnikiem „i”, czyli **&&.**
- Na osi Y piłka musi swoją dolną krawędzią dotykać górnej krawędzi paletki, czyli odległość piłki od górnej krawędzi

```
if (yPilki <= 0  || yPilki + srednicaPilki >= yp && xPilki > xp && xPilki < xp + szerPaletki){
    dy = dy*-1;
}
```

musi być co najmniej taka, jak odległość paletki od górnej krawędzi okna, pomniejszona o średnicę piłki. Można to zapisać jako **yPilki + srednicaPilki >= yp**. Ten warunek musi być spełniony wtedy, kiedy są spełniane dwa warunki opisane w poprzednim punkcie – więc trzeba go z nimi połączyć spójnikiem „i".

Cały warunek w tej instrukcji powinien mieć postać: **yPilki <= 0 || yPilki + srednicaPilki >= yp && xPilki > xp && xPilki < xp + szerPaletki**.

Ruch paletki

By sprawdzić, czy piłka odbija się od paletki, konieczne będzie wprawienie paletki w ruch. W tym celu należy wprowadzić do programu mechanizm obsługi klawiatury – to za jej pomocą, a konkretnie klawiszami strzałek, będzie sterowana paletka.

1 Do klasy **PanelGry** musimy dopisać klasę wewnętrzną. Klasy wewnętrzne są użyteczne tylko w kontekście klasy, w której się znajdują. Gdyby klasa miała być użyteczna poza nią, powinna być poza nią zdefiniowana. Ale nowa klasa ma służyć jedynie do zmieniania współrzędnych paletki, które są dostępne wyłącznie w klasie **PanelGry**. Klasę wewnętrzną definiujemy w taki sposób, jak każdą inną klasę, tylko wewnątrz klasy zewnętrznej jak metody klasy. Trzeba stworzyć klasę dziedziczącą po klasie **KeyAdapter** – pozwala ona (a konkretnie jej metoda **keyPressed**) między innymi na odczyt, jaki klawisz został ostatnio naciśnięty. By dodać wewnętrzną klasę dziedziczącą po klasie **KeyAdapter** o nazwie **Sterowanie**, należy napisać **private class Sterowanie extends KeyAdapter { }**.

```
private class Sterowanie extends KeyAdapter {

}
```

2 By klasa bazowa **KeyAdapter** była rozpoznawana przez program, trzeba dokonać odpowiedniego importu – **import java.awt.event.KeyAdapter**.

3 Nowa klasa po **KeyAdapter** dziedziczy metodę **keyPressed** – jednak należy metodę tę nadpisać, by program w zależności od tego, który klawisz zostanie naciśnięty, zachowywał się tak, jak chce tego programista. Dlatego użyjemy zapisu **@Override** przed definicją metody **keyPressed**, która powinna mieć postać **public void keyPressed(KeyEvent e) { }**, gdzie pod **KeyEvent e** kryje się obiekt, w którym przechowywana będzie informacja o tym, który klawisz był ostatnio naciśnięty.

```
private class Sterowanie extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e)  {

    }
}
```

4 By klasa **KeyEvent** była rozpoznawana przez program, trzeba dokonać importu biblioteki – **import java.awt.event.KeyEvent;**.

```
import java.awt.Graphics;
import java.util.Random;
import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
```

5 Informacja o tym, który z klawiszy został naciśnięty, jest schowana pod zapisem

```
@Override
public void keyPressed(KeyEvent e) {
    int key = e.getKeyCode();

}
```

e.getKeyCode(). Ukrywa się tam liczba całkowita odpowiadająca swoim kodem jednemu z klawiszy klawiatury. Tę liczbę możemy zachować w zmiennej. W tym celu można utworzyć zmienną o nazwie **key** i jako wartość dać jej to, co zwraca polecenie, w którym przechowywana jest liczba odpowiadająca klawiszom. Powinno to mieć formę: **int key = e.getKeyCode();**

6 Czy aby sprawdzić jakiś klawisz, trzeba znać jego kod wyrażony liczbą? Niekoniecznie – należy jednak znać jego nazwę w klasie **KeyEvent** (i to też nie jest do końca konieczne, gdyż podczas pisania IDE może nam sugerować dostępne polecenia, wśród których są także nazwy klawiszy). Numer odpowiadający strzałce w lewą stronę kryje się pod **KeyEvent.VK_LEFT**. Poprzez instrukcję warunkową należy sprawdzić, czy zmienna **key** jest równa **KeyEvent.VK_LEFT** i wtedy zmienić współrzędną **x** paletki tak, by przesunęła się w lewą stronę. Warunek ten można połączyć z jeszcze jednym, który powinien być równocześnie spełniony. Drugi z warunków dotyczy sprawdzenia aktualnej odległości paletki od lewej krawędzi okna. Chodzi o to, żeby nie dało się przesunąć paletki poza okno gry. Aby można było przesunąć paletkę, musi ona być w oknie gry, a znajduje się w nim wtedy, kiedy jej współrzędna **x** jest większa od zera. Cała instrukcja warunkowa powinna mieć postać: **if (key == KeyEvent.VK_LEFT && xp > 0) { }**.

```
@Override
public void keyPressed(KeyEvent e) {
    int key = e.getKeyCode();

    if (key == KeyEvent.VK_LEFT && xp > 0) {

    }
}
```

7 Następnie należy określić, jaka instrukcja ma się wykonać, gdy warunki będą spełnione, a ma to być zmniejszenie wartości pola **xp**, co można zapisać następująco: **xp = xp - 10;**.

```
@Override
public void keyPressed(KeyEvent e) {
    int key = e.getKeyCode();

    if (key == KeyEvent.VK_LEFT && xp > 0) {
        xp = xp - 10;
    }
}
```

8 Podobnie powinno wyglądać przesuwanie paletki w prawą stronę. Trzeba zbudować instrukcję warunkową, w której muszą być spełnione dwa warunki – jeden mówiący o wciśnięciu klawisza strzałki w prawą stronę, a drugi o tym, że paletka swoją prawą krawędzią mieści się jeszcze w oknie gry. Cała instrukcja powinna mieć zatem formę: **if (key == KeyEvent.VK_RIGHT && xp < OKNO_SZER - szerPaletki) { }**.

```
@Override
public void keyPressed(KeyEvent e) {
    int key = e.getKeyCode();

    if (key == KeyEvent.VK_LEFT && xp > 0) {
        xp = xp - 10;
    }

    if (key == KeyEvent.VK_RIGHT && xp < OKNO_SZER - szerPaletki) {

    }
}
```

9 Gdy warunki z instrukcji będą spełnione, pole **xp** powinno zwiększyć swoją wartość o tyle, o ile zmniejszało się w przypadku spełnienia warunków z poprzedniej instrukcji

```
private class Sterowanie extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {
        int key = e.getKeyCode();

        if (key == KeyEvent.VK_LEFT && xp > 0) {
            xp = xp - 10;
        }

        if (key == KeyEvent.VK_RIGHT && xp < OKNO_SZER - szerPaletki) {
            xp = xp + 10;
        }
    }
}
```

warunkowej. Należy zatem napisać: **xp = xp + 10;**. W ten sposób zakończyliśmy definicję metody **keyPressed** – możemy zająć się jej wywoływaniem.

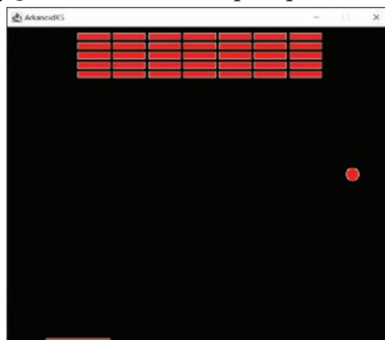
10 Wspomniana metoda będzie automatycznie wywoływana, gdy zostanie naciśnięty jakiś klawisz, jednak najpierw trzeba dodać obiekt klasy **Sterowanie** do klasy **PanelGry**. W tym celu w konstruktorze klasy **PanelGry** należy napisać **addKeyListener(new Sterowanie());**.

11 Niezbędne jest też wywołanie polecenia **setFocusable(true);**. Zdarzenia wysyłane są tylko do komponentu, który

```
    timer = new Timer(10, this);
    timer.start();
    addKeyListener(new Sterowanie());
    setFocusable(true);
}
```

jest aktywny. Można powiedzieć, że wspomniane polecenie „aktywuje” komponent (czyli w tym wypadku **PanelGry**).

12 Teraz zarówno piłka, jak i paletka są już w ruchu. Do zakończenia gry pozostaje jeszcze dodanie wykrywania kolizji pomiędzy piłką a cegłami i zakończenie gry, gdy gracz nie zdoła odbić piłki paletką.



```
public PanelGry() {
    initComponents();
    setBackground(Color.black);
    setPreferredSize(new Dimension(OKNO_SZER, OKNO_WYS));

    for (int i = 0; i < kolumny; i++) {
        for (int j = 0; j < wiersze; j++) {
            ceglyX[i][j] = i * (szerCegla + ceglyOdstep) + ceglyOdstepLewa;
            ceglyY[i][j] = j * (wysCegla + ceglyOdstep) + ceglaOdstepGora;
            czyZbite[i][j] = false;
        }
    }

    timer = new Timer(10, this);
    timer.start();
    addKeyListener(new Sterowanie());
```

Zderzenie piłki z cegłą

Do wykrywania, czy piłka uderzyła w cegłę, należy zbudować oddzielną metodę. Powinna ona mieć następującą postać, gdzie:

A – jest deklaracją metody o nazwie **zderzenieZCegla**;
B – jest pętlą **for**, która przechodzi po kolejnych kolumnach cegieł;
C – jest pętlą **for**, która przechodzi po kolejnych wierszach w kolumnie cegieł;
D – jest instrukcją warunkową sprawdzającą, czy cegła, do której odnosi się numer wyznaczony przez iteracje obu pętli, nie została jeszcze zbita;
E – jeśli warunek o oznaczeniu cegły jako jeszcze nie zbitej jest spełniony, kolejnym krokiem jest stworzenie instrukcji warunkowej do sprawdzania, czy piłka koliduje teraz z cegłą, do której odnoszą się zmienne **i** i **j**;
F – zestaw warunków, które muszą być jednocześnie spełnione, by program uznał, że piłka uderzyła w cegłę, lecąc do niej od dołu bądź od góry;
G – zmiana kierunku ruchu piłki na osi Y; zapis ten tożsamy jest z zapisem **dy = dy*-1**;
H – oznaczenie cegły, z którą koliduje piłka, jako zbitej;
I – **else if** pozwala na dodanie do instrukcji warunkowej kolejnego warunku, który będzie sprawdzony tylko w wypadku, gdy poprzednie warunki nie zostały spełnione;
J – zestaw warunków, które muszą być spełnione, by program uznał, że piłka uderzyła w cegłę, lecąc do niej z boku;
K – zmiana kierunku ruchu piłki na osi X; zapis ten jest tożsamy z zapisem **dx = dx*-1**;

```
@Override
public void actionPerformed(ActionEvent e) {
    zderzenieZCegla();
    ruchPilki();
    repaint();
}
```

Metodę **zderzenieZCegla** należy wywołać w metodzie wywoływanej przez timer. Sprawi to, że z każdym ruchem piłki będzie wykonywane też sprawdzenie, czy uderzyła ona w cegłę.

```
A private void zderzenieZCegla() {
  B for (int i = 0; i < kolumny; i++) {
    C for (int j = 0; j < wiersze; j++) {
      D if (czyZbite[i][j] == false) {
        E if (xPilki > ceglyX[i][j]
        F         && xPilki < ceglyX[i][j] + szerCegla
                  && yPilki + srednicaPilki > ceglyY[i][j]
                  && yPilki < ceglyY[i][j] + wysCegla) {
          G dy = -dy;
          H czyZbite[i][j] = true;
          } else if (yPilki > ceglyY[i][j]
          I       && yPilki < ceglyY[i][j] + wysCegla
              J   && xPilki + srednicaPilki > ceglyX[i][j]
                  && xPilki < ceglyX[i][j] + szerCegla) {
          K dx = -dx;
          H czyZbite[i][j] = true;
          }
        }
      }
    }
}
```

Zakończenie gry

Zakończenie gry może mieć dwa warianty – wygraną i przegraną. Pierwsza z opcji występuje, gdy zostaną zbite wszystkie cegły, druga zaś, gdy piłka wypadnie z okna gry poprzez dolną krawędź. Zakończenie gry powinno skutkować wyświetleniem odpowiedniego napisu i ukryciem tego, co dotychczas było widoczne. Będzie to wymagało edycji metody odpowiedzialnej za rysowanie na panelu.

1 Należy wprowadzić specjalne pole w klasie **PanelGry**, którego wartość będzie mówiła o tym, jaki jest stan gry. Deklarujemy pole **stanGry** jako typ danych **String**. Będzie ono mogło przyjmować wartości tekstowe – na przykład: „trwa", „wygrana", „przegrana". Należy napisać zatem **private String stanGry = "trwa";**.

```
private boolean czyZbite[][] = n
private int dx = 2;
private int dy = -2;
private Timer timer;
private String stanGry = "trwa";
```

2 By modyfikować wartość nowego pola klasy, można zbudować nową metodę, która będzie sprawdzała położenie piłki i liczbę cegieł do zbicia. Nowa metoda **sprawdzStan** może mieć postać, gdzie:

```
private void sprawdzStan(){
A  if (yPilki > OKNO_WYS)
   {
   B  stanGry = "PRZEGRYWASZ";
   }
C  int niezbite = 0;
D  for (int i = 0; i < kolumny; i++) {
   E  for (int j = 0; j < wiersze; j++) {
      F  if (czyZbite[i][j] == false) {
         G  niezbite += 1;
         }
      }
   }
H  if (niezbite == 0)
   {
   I  stanGry = "WYGRYWASZ";
   }
}
```

A – jest instrukcją warunkową sprawdzającą, czy piłka opuściła już okno gry przez jego dolną krawędź;
B – jeśli warunek o opuszczeniu okna przez piłkę został spełniony, ustawia wartość pola **stanGry** na **PRZEGRYWASZ** – napis ten zostanie dalej wykorzystany do wyświetlenia go na ekranie;
C – tworzy nową zmienną do zliczenia cegieł, które pozostały na ekranie;
D – jest pętlą **for**, która przechodzi po kolejnych kolumnach cegieł;
E – jest pętlą **for**, która przechodzi po kolejnych wierszach w kolumnie cegieł;
F – jest instrukcją warunkową, sprawdzającą, czy cegła, do której odnosi się numer wyznaczony przez iteracje obu pętli, nie została jeszcze zbita;
G – jeżeli cegła nie została zbita, zwiększa wartość zmiennej zliczającej cegły, które nie zostały jeszcze zbite;
H – instrukcja warunkowa sprawdzająca, czy policzono jakiekolwiek cegły, które nie zostały zbite;
I – jeśli brak cegieł do zbicia – ustawia wartość pola **stanGry** na **WYGRYWASZ**.

3 Metoda do sprawdzania stanu rozgrywki powinna być wywoływana wraz ze zdarzeniem timera w metodzie **actionPerformed**.

```
@Override
public void actionPerformed(ActionEvent e) {
    sprawdzStan();
    zderzenieZCegla();
    ruchPilki();
    repaint();
}
```

```
private void koniec(Graphics g){

}
```

4 Powinna zostać zdefiniowana metoda **koniec**, która będzie umieszczała na środku ekranu napis **WYGRYWASZ** lub **PRZEGRYWASZ** (w zależności od tego, jaką wartość ma **stanGry**).

```
private void koniec(Graphics g){
    Font czcionka = new Font("Helvetica", Font.BOLD, 14);
}
```

5 W metodzie **koniec** poprzez linię **Font czcionka = new Font(„Helvetica", Font.BOLD, 14);** należy utworzyć obiekt klasy **Font**, który posłuży jako czcionka do wygenerowania napisu. Liczba 14 jest tu rozmiarem czcionki, można oczywiście użyć innego rozmiaru, a także czcionki innej niż Helvetica.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.Font;
```

6 By klasa **Font** była rozpoznana przez program, należy dodać import biblioteki – **import java.awt.Font;**.

7 Dalej w metodzie **koniec** należy poprzez linię **FontMetrics metr = getFontMetrics(czcionka);** utworzyć obiekt, który pozwoli na obliczenie długości napisu tworzonego przez wcześniej zadeklarowaną czcionkę. Dzięki temu będzie możliwe obliczenie dokładnej lokalizacji napisu, tak by odległości od prawej i lewej krawędzi okna były takie same.

```
private void koniec(Graphics g){
    Font czcionka = new Font("Helvetica", Font.BOLD, 14);
    FontMetrics metr = getFontMetrics(czcionka);
}
```

```
import java.awt.Font;
import java.awt.FontMetrics;
```

8 By klasa **FontMetrics** była rozpoznana przez program, należy dodać import biblioteki: **import java.awt.FontMetrics;**.

9 Poleceniem **g.setColor(Color.white);** należy ustawić kolor tworzonego napisu (w naszym przykładzie jest to kolor biały).

```
private void koniec(Graphics g){
    Font czcionka = new Font("Helvetica", Font.BOLD, 14);
    FontMetrics metr = getFontMetrics(czcionka);
    g.setColor(Color.white);
}
```

10 Następnie poleceniem **g.setFont(czcionka);** przypisujemy wcześniej utworzony obiekt klasy **Font** jako czcionkę do tworzenia napisów.

```
private void koniec(Graphics g){
    Font czcionka = new Font("Helvetica", Font.BOLD, 14);
    FontMetrics metr = getFontMetrics(czcionka);
    g.setColor(Color.white);
    g.setFont(czcionka);
}
```

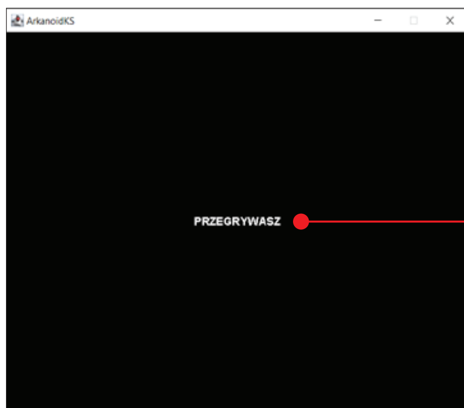
11 Poleceniem **g.drawString(stanGry, (OKNO_SZER - metr.stringWidth(stanGry)) / 2, OKNO_WYS / 2);** należy utworzyć na środku okna gry napis, którego tekst będzie dokładnie taki jak wartość pola **stanGry**.

```
private void koniec(Graphics g){
    Font czcionka = new Font("Helvetica", Font.BOLD, 14);
    FontMetrics metr = getFontMetrics(czcionka);
    g.setColor(Color.white);
    g.setFont(czcionka);
    g.drawString(stanGry, (OKNO_SZER - metr.stringWidth(stanGry)) / 2, OKNO_WYS / 2);
}
```

```
private void rysuj(Graphics g) {
    if (stanGry == "trwa"){
        rysujPaletke(g);
        rysujPilke(g);
        rysujCegly(g);
    }
}
```

12 Metoda **rysuj** powinna zostać zmodyfikowana w taki sposób, by obecne w niej do tej pory wywołania metod rysujących wykonywały się tylko wtedy, gdy pole **stanGry** ma wartość **trwa**. W przeciwnym wypadku powinna być wykonywana metoda **koniec**.

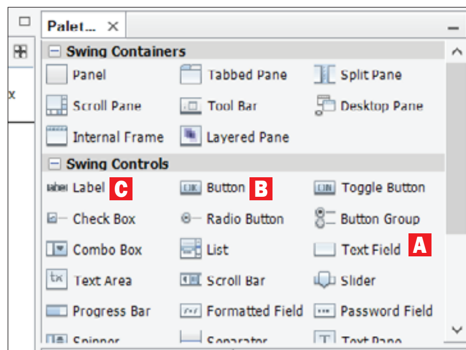
```
        rysujCegly(g);
    } else {
        koniec(g);
    }
}
```



Przetestujmy teraz grę – zależnie od wyniku, jaki uzyskamy, po rozgrywce w oknie gry powinien pojawić się odpowiedni napis.

Programy użytkowe: oblicz swoje BMI

Programowanie to oczywiście nie tylko tworzenie gier. To także tworzenie różnego rodzaju oprogramowania użytkowego. W tym także dobrze sprawdzi się Java. Jako przykład oprogramowania użytkowego mogą posłużyć różnego rodzaju przeliczniki, które pobierają od użytkownika dane wejściowe, dokonują obliczeń, a potem prezentują wyniki. Przykładem takiego programu może być narzędzie do obliczania BMI (Body Mass Index) – współczynnika masy ciała. Użytkownik podaje wzrost i wagę, a program, korzystając ze wzoru, zwraca mu wartość współczynnika.

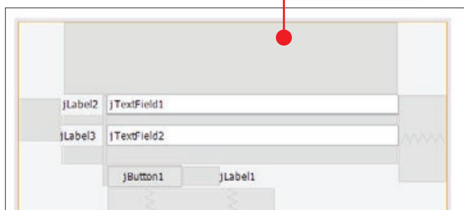


Program do obliczania BMI

1 Tworzymy projekt zgodnie z instrukcją z rozdziału drugiego, dodając do niego okno, czyli **JFrame Form**. Na tej formatce można zbudować wygląd aplikacji – w programach użytkowych ważną rolę pełnią przyciski i pola tekstowe.

2 Elementy okna programu przeciągamy na formatkę z panelu po prawej stronie. Program do obliczania BMI powinien składać się z: dwóch pól tekstowych (**Text Field** **A**) do wpisania wzrostu i wagi; przycisku (**Button** **B**) do uruchomienia obliczania; trzech etykiet (**Label** **C**) – jednej do prezentacji wyniku, a dwóch do podpisania pól tekstowych.

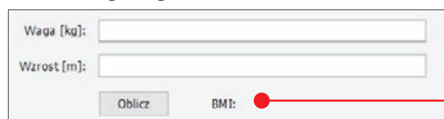
3 Przeciągnięte elementy układamy w sposób podobny do zaprezentowanego.



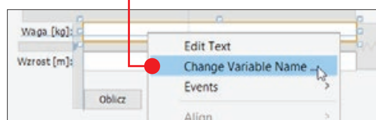
```
private void btnObliczActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}
```

4 By nadać elementom odpowiadającą nam formę, można zmienić ich rozmiary (zaznaczając element, można regulować jego wymiary, rozciągając go). Ważne dla ostatecznego wyglądu programu są też napisy na jego poszczególnych elementach. Możemy je zmieniać poprzez zmianę wartości właściwości **text** w zakładce **Properties** na panelu pod możliwymi do wyboru elementami okna. Właściwości wyświetlają się zawsze dla aktualnie zaznaczonego elementu okna.

5 Pola tekstowe powinny mieć pusty **text**, przycisk powinien mieć słowo "Oblicz", a etykiety, zależnie od ich lokalizacji: "Wzrost [m]:", "Waga [kg]:", "BMI:" ●.



6 By ułatwić odnoszenie się do poszczególnych elementów okna programu z poziomu skryptu, można zmieniać ich standardowe nazwy. By to zrobić, klikamy prawym przyciskiem myszy na element, którego nazwę chcemy zmienić, i wybieramy opcję **Change Variable Name** ●, by wyświetlić pole do podania nowej nazwy. W naszym przykładzie pola tekstowe powinny nazywać się **tbxWzrost** i **tbxWaga**, przycisk **btnOblicz**, a etykieta do wyświetlania współczynnika – **lblBmi**. Przyjęło się, że w programowaniu nazwy składają się ze skrótu wskazującego na rodzaj elementu i części wskazującej na jego zastosowanie. (Ponieważ do etykiet umieszczonych obok pól tekstowych nie będziemy się odnosić z poziomu skryptu, zmiana ich nazw nie jest konieczna, jednak warto wiedzieć, jak to robić).



7 Klikamy dwukrotnie na przycisk z napisem **Oblicz** na formatce. Spowoduje to wygenerowanie metody ● i przeniesienie nas do skryptu. Ta nowa metoda będzie się wykonywać automatycznie w uruchomionym programie, gdy użytkownik kliknie na przycisk.

8 W nowej metodzie umieszczamy następujące polecenia ●, które kolejno oznaczają:

A – tworzy zmienną typu **float** o nazwie **waga**, do której jako wartość trafia przerobiony na liczbę tekst wpisany do pola tekstowego **tbxWaga**.

B – tworzy zmienną typu **float** o nazwie **wzrost**, do której jako wartość trafia przerobiony na liczbę tekst wpisany do pola tekstowego **tbxWzrost**.

C – tworzy zmienną o nazwie **bmi** – typu **float**, do której zgodnie ze wzorem na współczynnik BMI trafia wartość zmiennej **waga** podzielona przez podniesioną do kwadratu wartość zmiennej **wzrost**.

D – tworzy zmienną typu **String** o nazwie **napis**, w której przechowywany jest przerobiony z liczby na tekst współczynnik BMI.

E – współczynnik BMI przerobiony do formy tekstowej jest umieszczony na etykiecie **lblBmi**, przy czym przed jego wartością dopisywany jest tekst "BMI: ".

Uruchomiony program daje efekt ●.




```
private void btnObliczActionPerformed(java.awt.event.ActionEvent evt) {
A   float waga = Float.parseFloat(tbxWaga.getText());
B   float wzrost = Float.parseFloat(tbxWzrost.getText());
C   float bmi = waga/(wzrost*wzrost);
D   String napis = "BMI: "+ String.valueOf(bmi);
E   lblBmi.setText(napis);
}
```

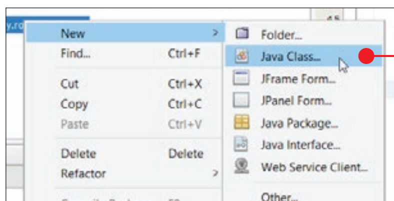
4 Automatyzacja pracy w Javie

Java pozwala na pisanie nie tylko aplikacji okienkowych czy konsolowych, ale także takich, które okna w ogóle nie mają. Tego typu programy mogą na przykład symulować działania użytkownika – to boty

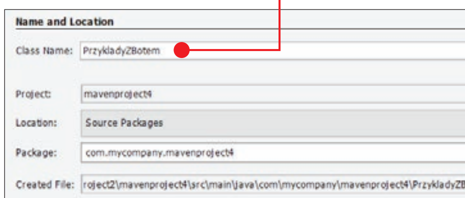
W tym rozdziale poznamy zestaw przydatnych poleceń wykorzystywanych do zautomatyzowania określonych czynności. Skrypty napisane w Javie mogą na przykład uruchamiać inne programy, sterować kursorem myszy czy symulować naciśnięcia klawiszy. Część tych funkcji można wykorzystać na przykład do budowy własnego bota, który będzie klikał w określony punkt ekranu (co jest przydatne w grach typu „kliker"). Innym zastosowaniem może być stworzenie programu, który będzie uruchamiał edytor tekstowy, a następnie sam wpisze w nim jakiś tekst. I to będzie pierwsze zadanie, którym się zajmiemy.

Bot piszący w Notatniku

By stworzyć bota piszącego w Notatniku, należy utworzyć projekt zgodnie z instrukcją z rozdziału **2**, jednak bez dodawania do niego **JFrame Form** – jest to okno programu, a nasz program nie potrzebuje okna.



1 Zamiast tego, do projektu należy dodać klasę – **Java Class**. Możemy nazwać tę klasę **PrzykladyZBotem**.



2 W przypadku gdy dodajemy do projektu jedynie plik z klasą, IDE wygeneruje początkowo tylko kilka komentarzy, nazwę paczki (**package**) i samą deklarację klasy. W pliku nie ma konstruktora czy metody **main**. Klasę można wykorzystać na wiele sposobów, a w wygenerowanym pliku znajduje się niezbędne minimum potrzebne do tego, by zacząć dalsze pisanie kodu.

Uruchomienie Notatnika poprzez skrypt

W przypadku naszego projektu niezbędne jest stworzenie metody **main** – w jej treści można zamieścić wszystkie zadania, które ma wykonać bot.

1 Wewnątrz klasy należy zadeklarować metodę, używając polecenia **public static void main(String[] args) { }**.

```
public class PrzykladyZBotem {
    public static void main(String[] args) {

    }
}
```

2 Poleceniem **String nazwa = „notepad.exe";** można utworzyć zmienną, w której przechowywana będzie nazwa

```
public class PrzykladyZBotem {
    public static void main(String[] args) {
        String nazwa = "notepad.exe";
```



```
public static void main(String[] args) throws IOException {
```

uruchamianego programu. Założeniem jest uruchomienie aplikacji wbudowanej w Windows – **Notatnika**. Nazwa pliku wykonywalnego tego programu to właśnie **notepad.exe**.

3 Poprzez polecenie **Runtime run = Runtime.getRuntime();** należy utworzyć obiekt **run** klasy **Runtime**. Obiekt ten pozwoli na wywołanie polecenia **exec**, które umożliwia uruchamianie aplikacji.

```
public static void main(String[] args) {
    String nazwa = "notepad.exe";
    Runtime run = Runtime.getRuntime();
```

4 Polecenia **exec** używamy w formie: **run.exec(nazwa);**.

```
13  public class PrzykladyZBotem {
14      public static void main(String[] args) {
15          String nazwa = "notepad.exe";
16          Runtime run = Runtime.getRuntime();
            run.exec(nazwa);
18      }
19  }
```

WYJĄTKI

Programowanie bardzo często opiera się na... błędach. Błędami mogą być literówki czy inne pomyłki wynikające z pisania. Błędy mogą też wynikać z niepoprawnych danych, jakie trafiają do programu. Takie błędy, których często nie sposób przewidzieć, nazywamy **wyjątkami**. Jednym ze sposobów zgłaszania wyjątków jest polecenie **throw**. Po nim podajemy, jaki wyjątek ma być zgłoszony. W nagłówku metody najpierw należy zadeklarować, jakie typy wyjątków dana metoda może zgłaszać, co robimy za pomocą instrukcji **throws**, wymieniając po przecinku różne ich rodzaje. Później w kodzie danej metody można je zgłosić, korzystając właśnie z instrukcji **throw**.

5 Podanie tego polecenia skutkuje wystąpieniem błędu. Klikamy na ikonę błędu, by rozwinąć sugerowane przez IDE rozwiązania, i wybieramy z listy pozycję **Add throws clause for java.io.IOException**.

```
13  public class PrzykladyZBotem {
14      public static void main(String
15          String nazwa = "notepad.ex
16          Runtime run = Runtime.getR
            run.exec(nazwa);
18
19
20
```

Add throws clause for java.io.IOException
Surround Statement with try-catch
Surround Block with try-catch
Assign Return Value To New Variable

6 Program modyfikuje nieco deklarację metody **main**, dodając do niej: **throws IOException**, ale też automatycznie uzupełnia **import** odpowiedniej biblioteki.

```
8   import java.io.IOException;
```

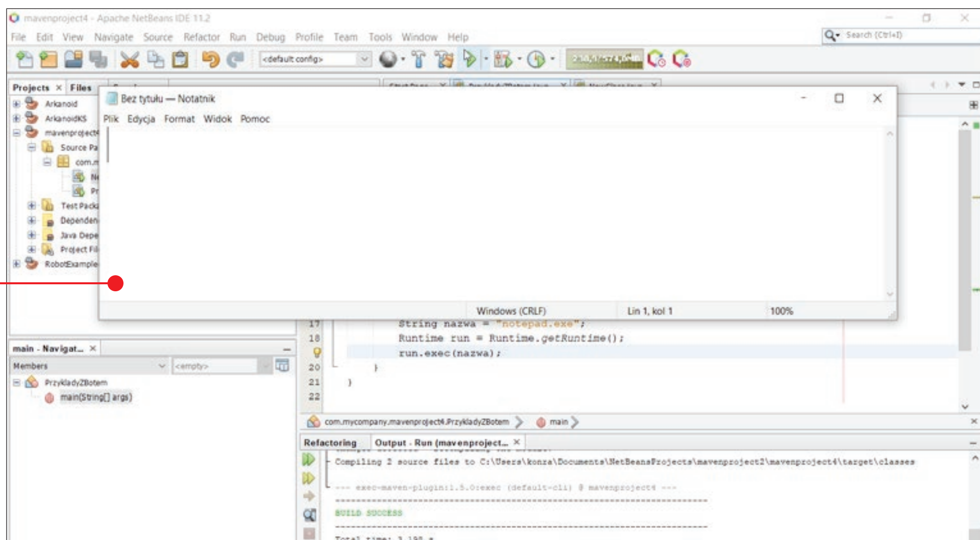
7 Jeśli teraz uruchomimy program (pamiętajmy, że pierwsze uruchomienie programu wymaga określenia głównej klasy z metodą **main** – w tym przypadku mamy tylko jedną klasę i metodę **main** w projekcie), po chwili powinien uruchomić się Notatnik.

Symulowanie wciskania klawiszy

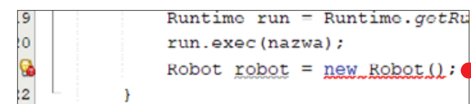
Mając uruchomiony Notatnik, można zająć się pisaniem w tym edytorze tekstowym. Program powinien mieć możliwość symulowania różnych zachowań użytkownika – w tym wciskania klawiszy.

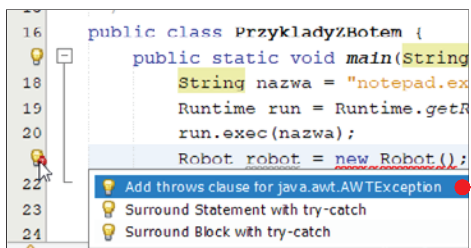
1 Java zawiera bibliotekę, która umożliwia takie operacje. By móc korzystać z tej biblioteki, trzeba ją importować. Można to zrobić za pomocą polecenia **import java.awt.Robot;**.

```
import java.io.IOException;
import java.awt.Robot;
```



2 Następnie w metodzie **main** można utworzyć obiekt klasy **Robot** – pisząc: **Robot robot = new Robot();** ●. Po wpisaniu takiej linii kodu IDE kolejny raz poinformuje o błędzie.





3 Rozwijamy listę sugerowanych możliwości i wybieramy z niej pierwszą opcję – **Add throws clause for java.awt.AWT Exception** ●. Ponownie zostanie nieco zmodyfikowana definicja metody **main**, a dzięki temu zostanie dodana możliwość obsługi wyjątków typu **AWTException** ●.

4 Dodany zostanie też import niezbędnej biblioteki **import java.awt.AWTException;** ●.

```
import java.awt.AWTException;
import java.io.IOException;
import java.awt.Robot;
```

5 Operacje wykonywane przez obiekt klasy **Robot** mogą mieć pewien interwał. Ten odstęp czasu możemy ustawić poprzez jego metodę **setAutoDelay** (która w parametrze przyjmuje czas wyrażony w milisekundach). By kolejne operacje symulowane przez obiekt wykonywały się co pół sekundy, należy użyć polecenia **robot.setAutoDelay(500);** ●.

```
String nazwa = "notepad.exe";
Runtime run = Runtime.getRuntime();
run.exec(nazwa);
Robot robot = new Robot();
robot.setAutoDelay(500);
```

6 Do symulowania wciśnięcia klawisza używa się metody **keyPress** dla obiektu klasy **Robot**. Metoda ta wymaga podania w parametrze kodu (numeru) klawisza,

```
public class PrzykladyZBotem {
    public static void main(String[] args) throws IOException, AWTException {
        String nazwa = "notepad.exe";
```

którego naciśnięcie ma symulować. Tak jak w przypadku Arkanoida z poprzedniego rozdziału i obsługi sterowania paletką poprzez strzałki – nie jest konieczna znajomość numerów klawiszy, które chcemy symulować. Można za to skorzystać z **KeyEvent** i za pomocą tego obiektu odnosić się do klawiszy poprzez ich nazwy. I tak na przykład klawisz W to **KeyEvent.VK_W**. Jednak najpierw, by **KeyEvent** było rozpoznawane przez program, należy dokonać odpowiedniego importu – **import java.awt.event.KeyEvent;**.

```
import java.awt.AWTException;
import java.io.IOException;
import java.awt.Robot;
import java.awt.event.KeyEvent;
```

7 Załóżmy, że program po uruchomieniu Notatnika ma wpisać w nim „jestem botem". By symulować wciśnięcie klawisza J, należy użyć polecenia **robot.keyPress-**

```
robot.keyPress(KeyEvent.VK_J);
robot.keyPress(KeyEvent.VK_E);
robot.keyPress(KeyEvent.VK_S);
robot.keyPress(KeyEvent.VK_T);
robot.keyPress(KeyEvent.VK_E);
robot.keyPress(KeyEvent.VK_M);
robot.keyPress(KeyEvent.VK_SPACE);
robot.keyPress(KeyEvent.VK_B);
robot.keyPress(KeyEvent.VK_O);
robot.keyPress(KeyEvent.VK_T);
robot.keyPress(KeyEvent.VK_E);
robot.keyPress(KeyEvent.VK_M);
```

(KeyEvent.VK_J);. W kolejnych liniach kodu należy zapisać symulowanie następnych liter i spacji – tak by powstał napis „jestem botem".



Uciekający kursor myszy

Innym zastosowaniem obiektu klasy **Robot** może być stworzenie programu, który będzie przemieszczał kursor myszy w losowe miejsca. Skrypt ten możemy dopisać do poprzedniego przykładu wykorzystania klasy **Robot**, dalej w metodzie **main**. Zakładamy więc, że obiekt **robot** klasy **Robot** już istnieje i można korzystać z jego metod.

1 Ponieważ uciekający kursor będzie przenosił się w losowe miejsca na ekranie, aby uzyskać nową lokalizację dla kursora, należy utworzyć obiekt klasy **Random**. Za pomocą jego metod będzie możliwe wygenerowanie losowej (pseudolosowej) pozycji dla kursora.

```
Random r = new Random();
```

2 By klasa **Random** była rozpoznawana przez program, należy importować odpowiednią bibliotekę.

```
import java.awt.AWTException;
import java.io.IOException;
import java.awt.Robot;
import java.awt.event.KeyEvent;
import java.util.Random;
```

3 Dalej należy poprzez polecenie **for (int i=0; i<100; i++){ }** utworzyć pętlę **for**, która pozwoli na stukrotną zmianę pozycji kursora myszy, gdy wewnątrz pętli znajdzie się odpowiednie polecenie.

```
Random r = new Random();
for (int i=0; i<100; i++){

}
```

4 Do wnętrza pętli trzeba wpisać polecenie **robot.mouseMove(r.nextInt(1200),r.nextInt(800));**, gdzie w parametrach uruchomiona jest metoda **nextInt** dla obiektu klasy **Random**, by wyznaczyć kolejno

```
Random r = new Random();
for (int i=0; i<100; i++){
  robot.mouseMove(r.nextInt(1200),r.nextInt(800));
}
```

współrzędną **x** i **y** nowej lokalizacji kursora myszy. Liczba **1200** to maksymalna możliwa do wygenerowania współrzędna **x** kursora, a **800** – współrzędna **y**. Możemy też użyć innych liczb, aby zmienić zakres generowania lokalizacji dla kursora.

5 Kursor myszy będzie zmieniał swoją pozycję co pół sekundy, gdyż w poprzedniej części programu użyto polecenia **robot.setAutoDelay(500);** – jeśli chcemy, by lokalizacja kursora zmieniała się częściej, użyjmy ponownie polecenia **setAutoDelay** bezpośrednio przed skryptem napisanym w tej części rozdziału i podajmy jako parametr oczekiwany czas przerwy między kolejnymi zmianami lokalizacji kursora.

6 Program po wykonaniu napisu w Notatniku powinien teraz co pół sekundy przemieszczać kursor myszy w nowe miejsce.

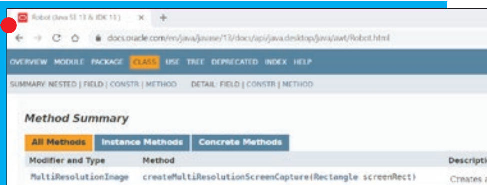
Przydatne metody klasy Robot

Klasa **Robot** to nie tylko metody zaprezentowane w przykładach. Obiekty tej klasy mogą robić też inne przydatne rzeczy. Wśród możliwości klasy **Robot** jest też tworzenie zrzutów ekranu czy sprawdzanie koloru piksela na ekranie w określonych koordynatach. To sprawia, że klasa **Robot** jest idealnym narzędziem do budowy botów. By lepiej się z nim zapoznać, możemy spróbować zbudować bota do gry typu „kliker" (polegającej na klikaniu na określony przycisk w celu zdobywania punktów).

METODA	OPIS
`delay(int ms)`	Pozwala na zatrzymanie programu na określony w milisekundach czas podany w parametrze metody. W przeciwieństwie do **setAutoDelay** wykonuje się jednorazowo, a nie po każdej czynności.
`keyRelease(int keycode)`	Analogicznie do **keyPress** – symuluje puszczenie określonego klawisza.
`mousePress(int buttons)`	Symuluje wciśnięcie przycisku myszy. Dla lewego przycisku myszy należy uruchomić ją z parametrem **InputEvent.BUTTON1_MASK**.
`mouseWheel(int wheelAmt)`	Symuluje kręcenie kółkiem myszy – parametr oznacza liczbę przekręceń. Liczba ujemna oznacza kręcenie w górę, a dodatnia w dół.

DOKUMENTACJA DO JAVY

By wykorzystać pełnię możliwości, jakie dają nam różnego rodzaju biblioteki, najlepiej zapoznać się z ich dokumentacją. Portal, gdzie znajduje się dokumentacja do bibliotek Java, znajduje się pod adresem **docs.oracle.com/en/java**. Z łatwością wyszukamy tam także dokumentację do biblioteki **java.awt.Robot**. Dzięki niej poznamy więcej możliwości tej biblioteki.



5 Podstawy tworzenia projektów 3D w Javie

Często główną przyczyną podjęcia decyzji o rozpoczęciu nauki programowania jest chęć tworzenia gier z grafiką 3D. Z tego rozdziału dowiemy się, jakie kroki podjąć, by móc tworzyć takie projekty, znając Javę

Minecraft

Minecraft jest grą o bardzo prostym założeniu. Oto gracz zostaje umieszczony w losowo generowanym świecie, który zbudowany jest z sześcianów, mających różne tekstury. Gracz może niszczyć bloki, stawiać nowe, atakować napotkane istoty, zbierać surowce czy wytwarzać przedmioty. Z kolejnymi wersjami gry jej możliwości rosną, a produkcja już teraz mocno różni się od wersji zaprezentowanej podczas premiery w 2011 roku.



Scena z gry Minecraft

Twórca gry Minecraft, Markus Persson „Notch”, rozpoczął pracę nad produkcją 10 maja 2009 roku. Już tydzień później ukazała się otwarta wersja testowa projektu. Do oficjalnej premiery w 2011 roku zespół programistów pracujących nad grą Minecraft rozrósł się, w międzyczasie powstała także firma Mojang AB, która oficjalnie odpowiadała za Minecrafta.

Minecraft odniósł ogromny sukces, na stałe znajdując miejsce w świadomości mnóstwa ludzi, szczególnie wśród dzieci. Na fali popularności gry na rynku zaczęły się pojawiać różnego rodzaju produkty powiązane z jej światem. I tak Minecraft to teraz także między innymi seria klocków LEGO, zabawki, seriale, książki i magazyny. Firma Mojang AB została przejęta przez prawdziwego giganta IT – firmę Microsoft. Wartość transakcji opiewała na 2,5 miliarda dolarów. Ta ogromna kwota robi wrażenie. Nic dziwnego, że wielu fanów tej produkcji chciałoby także stworzyć taką grę i odnieść podobny sukces.

Dlaczego w książce o Javie pojawia się historia Minecrafta? Dlatego że właśnie w tym języku programowania został stworzony Minecraft.

LWJGL 2

A konkretnie wykorzystano do tego **LWJGL**, czyli Light Weight Java Game Library. To zewnętrzna biblioteka przeznaczona do tworzenia gier. W przypadku Minecrafta zastosowano ją do wyświetlania grafiki, odtwarzania dźwięku oraz obsługi urządzeń wejściowych (klawiatura, mysz itp.). LWJGL łączy w sobie między innymi zastosowania do celów graficznych (OpenGL, Vulkan), dźwiękowych (OpenAL) i obliczeń równoległych (OpenCL). Do tworzenia Minecrafta wykorzystano LWJGL 2, jednak obecnie na rynku jest dostępna nowsza wersja tej biblioteki z gałęzi LWJGL 3.

LWJGL 3

Obecnie biblioteka LWJGL jest dostępna w wersji 3.2.3. By użyć tej biblioteki w swoich skryptach, należy pobrać ją ze strony o adresie **www.lwjgl.org** i dołączyć do projektu.

Aby umożliwić zapoznanie się z działaniem biblioteki i sposobem korzystania z niej, twórcy przygotowali przykładowe skrypty szkoleniowe. Znajdziemy je pod adresem **www.lwjgl.org/guide**.

Jeden z przykładów to składający się z wielu linijek kod, którego działanie polega na stworzeniu okna gry i wypełnieniu go czerwonym kolorem.

Twórcy przygotowali też zestaw projektów bardziej zaawansowanych – dema pokazujące działanie biblioteki w praktyce. Możemy je pobrać także ze strony www.lwjgl.org/guide, przechodząc nieco poniżej przedstawionego skryptu.

Get started with LWJGL 3 - LWJG
lwjgl.org/guide

```
import org.lwjgl.*;
import org.lwjgl.glfw.*;
import org.lwjgl.opengl.*;
import org.lwjgl.system.*;

import java.nio.*;

import static org.lwjgl.glfw.Callbacks.*;
import static org.lwjgl.glfw.GLFW.*;
import static org.lwjgl.opengl.GL11.*;
import static org.lwjgl.system.MemoryStack.*;
import static org.lwjgl.system.MemoryUtil.*;

public class HelloWorld {

    // The window handle
    private long window;

    public void run() {
        System.out.println("Hello LWJGL " + Version.getVersion() + "!");

        init();
        loop();

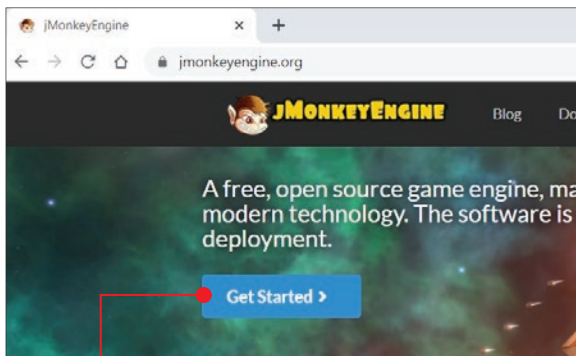
        // Free the window callbacks and destroy the window
        glfwFreeCallbacks(window);
        glfwDestroyWindow(window);

        // Terminate GLFW and free the error callback
        glfwTerminate();
        glfwSetErrorCallback(null).free();
    }

    private void init() {
        // Setup an error callback. The default implementation
        // will print the error message in System.err.
```

Silnik gier jMonkeyEngine

W programowaniu dąży się do maksymalnego uproszczenia procesu kodowania. W przypadku LWJGL kodowanie jest dość skomplikowane. Skrypty są długie i na pierwszy rzut oka trudno je zrozumieć. Dużo jest tam subtelności i każdy drobiazg może mieć duże znaczenie. W celu uproszczenia pracy, na przykład z projektami 3D, powstaje wiele silników gier. To narzędzia oparte na językach programowania i wprowadzające wiele mechanizmów właściwych dla konkretnego języka, jednak ułatwiających tworzenie rozbudowanych projektów.
Jednym z takich silników jest **jMonkeyEngine**. Narzędzie to oparto właśnie na LWJGL, jednak by z niego korzystać, nie trzeba znać zawiłych mechanizmów tej rozbudowanej biblioteki.



1 By rozpocząć pracę z tym silnikiem, należy wejść na stronę **https://jmonkeyengine.org** i kliknąć na przycisk **Get Started**.

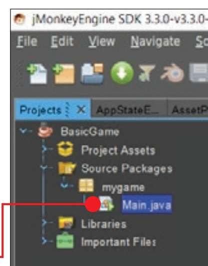
2 Następnie, klikając na przycisk **Download the SDK** **A**, pobieramy narzędzie wzorowane na NetBeans IDE, jednak zawierające w sobie silnik jMonkeyEngine.

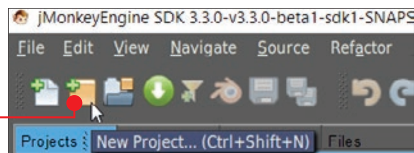
3 Po instalacji pobranego narzędzia klikamy na przycisk **New Project**, by utworzyć nowy projekt.

4 W nowo otwartym oknie, w sekcji **Categories**, wybieramy **JME3** **B**, a w sekcji **Projects** klikamy na **BasicGame** **C**. Zatwierdzamy wszystko, klikając na **Next** **D**.

5 W kolejnym kroku możemy określić nazwę nowego projektu oraz lokalizację **E**, w której zostanie zapisany. Projekt zostanie utworzony, gdy klikniemy na **Finish**.

6 By zapoznać się z początkową, wygenerowaną przez program treścią projektu, musimy rozwinąć drzewo z pakietem – jak w NetBeans IDE – by dostać się do pliku **Main.java**.





JMONKEYENGINE Blog Documentation License Community Store Github Discord

Quick Start

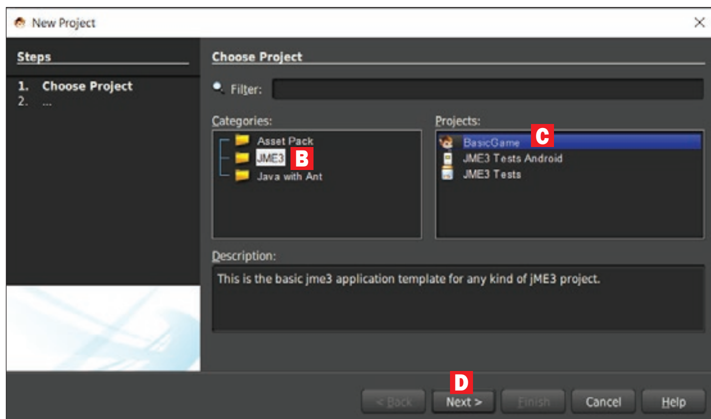
There are a variety of ways to use jMonkeyEngine:

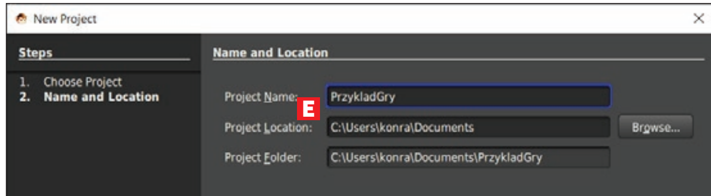
- Download the SDK
- Download the engine
- Add the libraries to a build script

Using the Netbeans-based SDK is by far the quickest solution to get you up and running. Everything needed is provided, along with extra tools and integrations, and is generally the place most users start their endevour. Download the SDK.

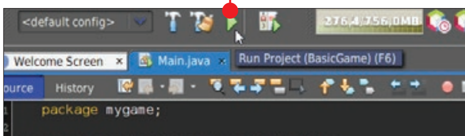
A

7 W pliku znajduje się baza projektu z grafiką 3D. Wygenerowany skrypt zawiera opis tworzenia niebieskiego sześcianu. Uruchamiamy program, klikając na przycisk **Run project**.

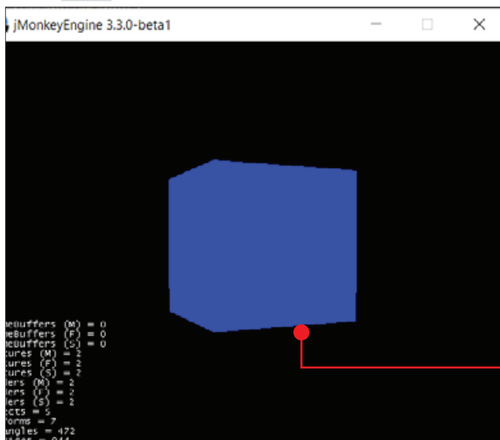




8 Okno graficzne aplikacji stworzonej przez silnik jMonkeyEngine nie od razu pokazuje nam to, co zostało zapisane w widocznym skrypcie. Najpierw wyświetlony jest ekran z informacją o użytym silniku, ekran ten pozwala także na dokonanie zmian dotyczących grafiki, między innymi rozdzielczości, korekcji



Gamma czy głębi koloru. By przejść do głównej treści programu, klikamy na przycisk **Continue**.





9 Właściwe okno aplikacji ze skryptu zawiera już wspomniany niebieski sześcian. Możemy oglądać go z różnych stron. Widokiem można sterować z wykorzystaniem kursora myszy, a także klawiszy W, A, S, D.

Welcome Screen | Main.java

Source | History

```
package mygame;

import com.jme3.app.SimpleApplication;
import com.jme3.material.Material;
import com.jme3.math.ColorRGBA;
import com.jme3.renderer.RenderManager;
import com.jme3.scene.Geometry;
import com.jme3.scene.shape.Box;

/**
 * This is the Main Class of your Game. You should only do initialization here.
 * Move your Logic into AppStates or Controls
 * @author normenhansen
 */
public class Main extends SimpleApplication {

    public static void main(String[] args) {
        Main app = new Main();
        app.start();
    }

    @Override
    public void simpleInitApp() {
        Box b = new Box(1, 1, 1);
        Geometry geom = new Geometry("Box", b);

        Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
        mat.setColor("Color", ColorRGBA.Blue);
        geom.setMaterial(mat);

        rootNode.attachChild(geom);
    }

    @Override
    public void simpleUpdate(float tpf) {
        //TODO: add update code
    }

    @Override
    public void simpleRender(RenderManager rm) {
        //TODO: add render code
    }
}
```

10 Przyjrzyjmy się jeszcze treści pliku **Main.java**. Po jego otwarciu zobaczymy krótki skrypt ●, którego zadaniem jest wyświetlenie w oknie programu niebieskiego sześcianu. Klasa **Main** dziedziczy po klasie **SimpleApplication**, która jest bazą dla projektów w jMonkeyEngine (patrz też strona 64).

Jak samodzielnie tworzyć obiekty 3D

Bazując na tym, co dostarcza startowy projekt jMonkeyEngine, można zaprojektować bardziej rozbudowaną scenę. Takich sześcianów może być więcej – zobaczmy, jak je stworzyć.

1 Do metody **simpleInitApp** dopisujemy linijkę **Geometry geom2 = new Geometry("Box", b);** **A**. Utworzymy w ten sposób nową geometrię bazującą na wcześniej utworzonym sześcianie. W ten sposób możliwe będzie dodanie w oknie projektu kolejnej kostki o tych samych wymiarach, jednak z przypisanym innym materiałem – będzie ona miała inny kolor. To właśnie materiał nadaje wygląd obiektom. Bez pokrycia obiektu materiałem nie byłoby go widać.

2 Poleceniem **Material mat2 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");** **B** tworzymy nowy materiał.

```
public void simpleInitApp() {
    Box b = new Box(1, 1, 1);
    Geometry geom = new Geometry("Box", b);
    Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
    mat.setColor("Color", ColorRGBA.Blue);
    geom.setMaterial(mat);

    rootNode.attachChild(geom);

    Geometry geom2 = new Geometry("Box", b);
```
A

```
Geometry geom2 = new Geometry("Box", b);
Material mat2 - new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
```
B

```
Geometry geom2 - new Geometry("Box", b);
Material mat2 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
mat2.setColor("Color", ColorRGBA.Red);
```
C

3 By materiał nie był „pusty”, należy przypisać do niego kolor, co można zrobić poleceniem **mat2.setColor(”Color”, ColorRGBA.Red);** C. W przykładowym poleceniu przypisywanym kolorem jest czerwony, jeśli

```
Geometry geom2 = new Geometry("Box", b);
Material mat2 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
mat2.setColor("Color", ColorRGBA.Red);
geom2.setMaterial(mat2);
rootNode.attachChild(geom2);
```
D

```
Geometry geom2 = new Geometry("Box", b);
Material mat2 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
mat2.setColor("Color", ColorRGBA.Red);
geom2.setMaterial(mat2);
rootNode.attachChild(geom2);
```
E

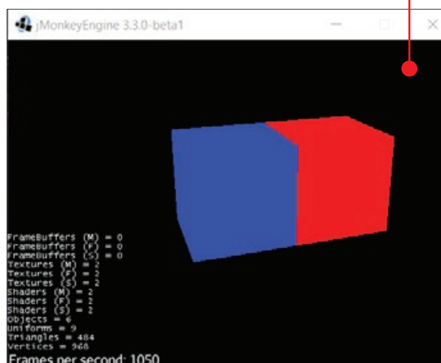
w miejsce **Red** podamy inną angielską nazwę koloru, to ten właśnie kolor zostanie ustawiony w materiale.

4 Dopisujemy jeszcze **geom2.setMaterial(mat2);** D, by utworzona wcześniej geometria została pokryta nowym materiałem.

5 Poprzez polecenie **rootNode.attachChild(geom2);** E dodajemy czerwoną kostkę do grafu sceny.

6 Gdy dodajemy obiekty do grafu sceny, domyślnie trafiają one w jedną lokalizację na środku układu współrzędnych. By dwie kostki – niebieska i czerwona – nie znajdowały się w tej samej lokalizacji, trzeba przesunąć jedną z nich. Można to zrobić poleceniem **geom2.move(2,0,0);** F (patrz kolejna strona) – wtedy czerwona kostka przesunie się o dwie jednostki na osi **X** (co oznacza przesunięcie w prawą stronę). Środkowa liczba w parametrze metody **move** mówi o przesunięciu na osi **Y** (góra-dół), a ostatnia – na osi **Z** (przód-tył).

7 Po uruchomieniu projektu zobaczymy dwa sześciany w różnych kolorach.



```
public void simpleInitApp() {
    Box b = new Box(1, 1, 1);
    Geometry geom = new Geometry("Box", b);
    Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
    mat.setColor("Color", ColorRGBA.Blue);
    geom.setMaterial(mat);

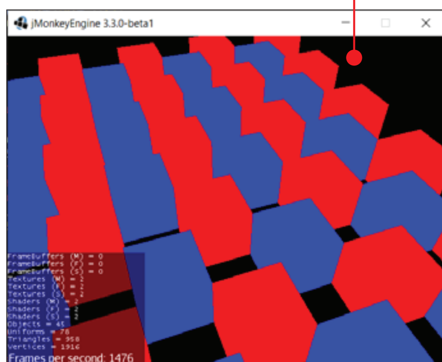
    rootNode.attachChild(geom);

    Geometry geom2 = new Geometry("Box", b);
    Material mat2 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
    mat2.setColor("Color", ColorRGBA.Red);
    geom2.setMaterial(mat2);
    rootNode.attachChild(geom2);
    geom2.move(2,0,0);
}
```

F

Jeszcze więcej sześcianów

W ramach ćwiczenia możemy spróbować tworzyć graficzne kostki w pętli, a nawet w pętlach – by było ich więcej. Zaprezentowany efekt można uzyskać, doprowadzając kod do następującej postaci. Tworzenie kostek można też zapętlać na inne sposoby, by osiągnąć jeszcze inne efekty.



Klasa SimpleApplication

Klasa **SimpleApplication** zawiera wiele ciekawych mechanizmów. To ona odpowiada za pierwszoosobowy widok i możliwość sterowania kamerą; dostarcza graf sceny umożliwiający zarządzanie renderowanymi modelami 3D oraz mapowanie wejścia w celu sterowania kamerą. By lepiej zapoznać się z działaniem tej klasy, warto przyjrzeć się znajdującym się w niej polom i metodom opisanym w tabelce obok. Nie są to jedyne pola i metody, jakie zawiera klasa **SimpleApplication**, bo ona sama dziedziczy sporo zawartości po klasie **Application**.

Mapowanie wejścia

Klasa dziedzicząca po **SimpleApplication** ma w sobie wbudowaną obsługę wielu klawiszy i myszy. Za co odpowiadają poszczególne klawisze? Część z nich służy do sterowania kamerą. Sterować kamerą można także myszą – przesuwając kursor. Przybliżanie i oddalanie się można zrealizować poprzez przekręcanie kółka myszy.

```
public void simpleInitApp() {
    Box b = new Box(1, 1, 1);
    for (int i = 0; i < 10; i = i + 2) {
        for (int j = 0; j < 10; j = j + 2) {
            Geometry geom = new Geometry("Box", b);
            Material mat = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
            mat.setColor("Color", ColorRGBA.Blue);
            geom.setMaterial(mat);

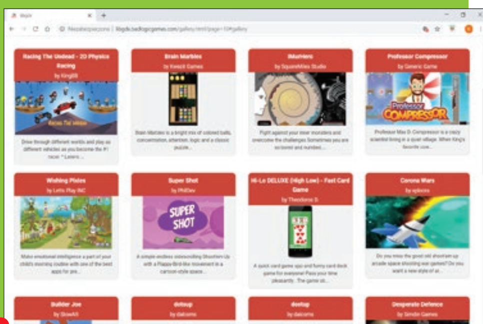
            rootNode.attachChild(geom);

            Geometry geom2 = new Geometry("Box", b);
            Material mat2 = new Material(assetManager, "Common/MatDefs/Misc/Unshaded.j3md");
            mat2.setColor("Color", ColorRGBA.Red);
            geom2.setMaterial(mat2);
            rootNode.attachChild(geom2);
            geom2.move((i * 2) + 2, 0, (j * 2) + 2);
            geom.move(i * 2, 0, j * 2);
        }
    }
}
```

NAZWA POLA/METODY	OPIS
rootNode	To pole klasy będące grafem elementów sceny. Przyjrzyjmy się metodzie **simpleInitApp** – w niej, dla obiektu rootNode, uruchamiana jest metoda **attachChild**. Utworzony wcześniej niebieski sześcian za pomocą tej metody jest dodawany do grafu elementów sceny.
guiNode	To kolejne pole klasy, które służy do dodawania elementów okna gry. W tym wypadku są to jednak elementy płaskie, nietrójwymiarowe, takie jak ikony, napisy czy paski.
flyCam	Dzięki temu polu klasa obsługuje kamerę, co daje możliwość obserwacji sześcianu z różnych perspektyw.
loadStatsView()	To metoda, która pozwala na umieszczenie w oknie projektu statystyk dotyczących działania aplikacji. Jest to szczególnie przydatne podczas tworzenia i sprawdzania działania aplikacji, a nieco mniej, gdy już gotowy program trafia do użytkownika.
loadFPSText()	Metoda, której wywołanie umieszcza na ekranie bieżącą liczbę fps (klatek na sekundę).
setDisplayFps(false)	Bieżąca liczba fps (klatek na sekundę) jest domyślnie widoczna na ekranie, mimo że w skrypcie, jaki silnik udostępnia programiście na start pracy nad nową grą, nie ma bezpośredniego wywołania metod za to odpowiedzialnych. Użycie metody **setDisplayFps** z parametrem **false** powoduje ukrycie tej domyślnie widocznej wartości.
setDisplayStatView(false)	Analogicznie jak w wypadku poprzedniej metody – w ten sam sposób można ukryć wszystkie statystyki (poza fps, które wymaga oddzielnego polecenia) widoczne domyślnie w oknie gry.
simpleInitApp()	To metoda, którą w implementacji klasy dziedziczącej po SimpleApplication należy nadpisać – to właśnie w niej tworzy się elementy okna i umieszcza je na scenie (dodaje do grafu sceny). Metoda ta jest nadpisana w przykładowym skrypcie i to właśnie w jej treści utworzony jest niebieski sześcian.
simpleUpdate(float tpf)	Kolejna metoda, którą trzeba nadpisać. Wszystkie operacje w niej umieszczone są wykonywane w pętli. Częstotliwość ich wykonywania wpisuje się w nawiasie w formie parametru.
simpleRender(RenderManager rm)	Metoda, którą również należy nadpisać. Pozwala ona na modyfikowanie bufora ramki (to część pamięci RAM) i jest przeznaczona do wykorzystywania przez doświadczonych programistów.

ALTERNATYWA DLA JMONKEYENGINE – LIBGDX

Innym popularnym narzędziem do tworzenia gier 3D w Javie jest **libGDX**. Framework ten jest wykorzystywany zarówno przez wielu niezależnych programistów, jak i przez największe korporacje IT, w tym Google (przykładem jest tu gra z gatunku MMO – **Ingress**). Oceniając możliwości silnika gier, najlepiej sprawdzić, jakie efekty da się w nim osiągnąć. Co można zrobić w libGDX, zobaczymy pod adresem **https://libgdx.badlogicgames.com/gallery.html**.



6 Java na Androida

Jedną z głównych przyczyn popularności Javy jest to, że wykorzystuje się ją do tworzenia aplikacji na system operacyjny Android. W tym rozdziale poznamy podstawy tworzenia aplikacji mobilnych

Android Studio

Mówiąc o tworzeniu aplikacji na Androida, należy zwrócić uwagę na oprogramowanie, jakie możemy w tym celu wykorzystać. Firma Google zaproponowała programistom dedykowane rozwiązanie. To program **Android Studio** (DVD-KOD: 001/002 (32-/64-BIT)). Nie oznacza to, że nie istnieją dla niego alternatywy. Aplikacje na Androida można tworzyć także poprzez inne programy, jak **NetBeans** (DVD-KOD: 003) czy **Eclipse** (DVD-KOD: 006) – wymaga to jednak odpowiedniego przygotowania tych narzędzi.
Jednak do pierwszego kontaktu z programowaniem na Androida najlepiej sprawdzi się dedykowane środowisko, dlatego wskazówki przedstawione w tym rozdziale zostały oparte na programie Android Studio, który można zainstalować z płyty dołączonej do książki.

KOTLIN

W 2019 roku język programowania Kotlin zastąpił Javę jako preferowany przez Google język tworzenia aplikacji na Androida. **Kotlin** – działający na maszynie wirtualnej Javy – został zaprojektowany jako przemysłowy, obiektowy język w pełni interoperacyjny z kodem napisanym w Javie, pozwalając firmom na stopniową migrację bazy kodu z Javy do Kotlina.

Android SDK

Tak jak do programowania w samej Javie potrzebne jest **JDK**, tak do programowania dla Androida należy zaopatrzyć się w **Android SDK**. Co ważne, jest ono instalowane wraz z Android Studio. Android SDK składa się z dwóch części: **SDK Tools** – wymaganej do tworzenia aplikacji niezależnie od wersji Androida, oraz **Platform Tools**, czyli narzędzi zmodyfikowanych pod kątem konkretnych wersji systemu.

W skład tej pierwszej części wchodzą między innymi:

- **android** – pozwala zarządzać wirtualnymi maszynami (**AVD Manager**), projektami, modułami SDK (**Android SDK Manager**),
- **ddms** (skrót od Dalvik Debug Monitor Server) – debugger (nazwa ta oznacza program komputerowy służący do dynamicznej analizy innych programów w celu odnalezienia i identyfikacji zawartych w nich błędów, które z angielskiego nazywane są bug, co można przetłumaczyć jako robak),
- **emulator** – emulator urządzenia z Androidem, którego można użyć do testowania stworzonych aplikacji pod różnymi wersjami systemu; tworzy on wirtualne środowisko Androida na komputerze,
- **sqlite3** – pozwala uzyskać dostęp do plików baz danych SQLite,
- **layoutopt** – analizuje layout (wygląd, rozmieszczenie elementów) aplikacji w celu zoptymalizowania ich pod kątem wydajności,
- **ProGuard** – zmniejsza, optymalizuje i zaciemnia kod poprzez usuwanie nieużywanych fragmentów oraz zmianę nazw klas, metod i pól,
- **mksdcard** – pozwala utworzyć obraz dysku możliwego do użycia z emulatorem, co ma symulować obecności zewnętrznej pamięci (przykładowo karty SD),
- **traceview** – graficzna przeglądarka logów wykonania aplikacji; log jest chronologicznym zapisem zdarzeń i działań użytkownika programu; takie pliki są pomocne w zlokalizowaniu problemów z oprogramowaniem zgłaszanych przez użytkowników,
- **zipalign** – optymalizuje pliki APK – Android Package Kit to format pliku będący odmianą formatu JAR, używany do dystrybucji i instalacji pakietów na system operacyjny Android.

W drugiej części znajdują się:

- **Android Debug Bridge** – wszechstronne narzędzie, które pozwala na kontrolowanie emulatora lub urządzenia z Androidem, na którym testowane są projekty,
- **fastboot** – program umożliwiający takie operacje, jak instalacja nowszej wersji Androida lub nawet innego systemu, zarządzanie partycjami czy odblokowywanie bootloadera.

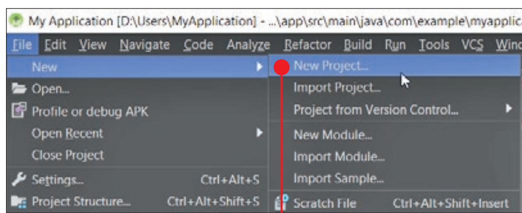
Używanie jakiegokolwiek IDE do programowania w Javie dla Androida nie jest obowiązkowe. Można edytować pliki tekstowe w dowolnym edytorze, a później budować i debugować aplikację za pomocą konsolowych narzędzi dostarczanych z SDK – analogicznie jak w wypadku JDK.

Activities – aktywności

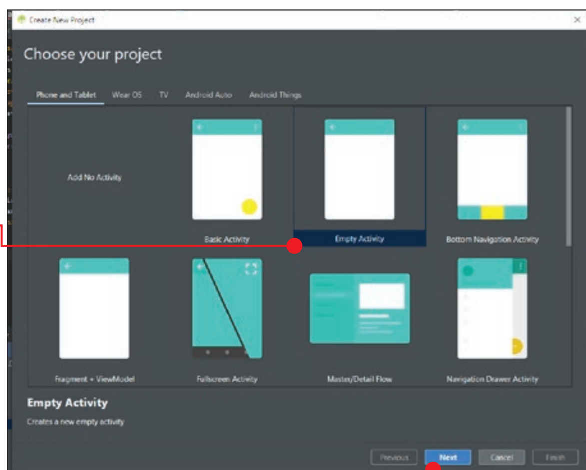
Activities to po polsku **aktywności**. Mocno upraszczając, aktywności są wykorzystywane do interakcji programu z użytkownikiem. A taka interakcja następuje poprzez okno aplikacji. I takie właśnie poszczególne okna programu nazywa się aktywnościami. Dzięki ich zastosowaniu możliwa jest optymalizacja programu – nie wczytuje się wszystkich jego elementów jednocześnie, ale kolejno, dopiero wtedy, gdy trzeba je wyświetlić. Ważnym elementem zastosowania aktywności jest też możliwość cofania się do poprzednich okien z zachowaniem ich stanu.

Jak stworzyć projekt

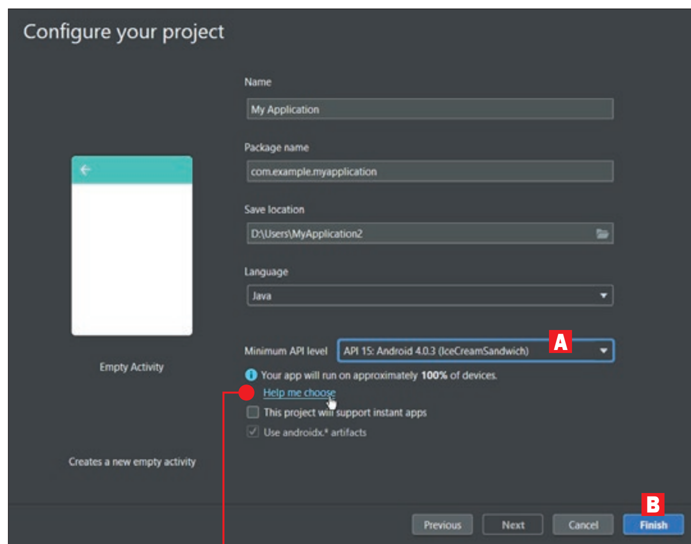
Mając już zainstalowane Android Studio, można przejść do utworzenia pierwszego projektu.



1 Po uruchomieniu narzędzia rozwijamy z paska menu pozycję **File** i wybieramy **New**, a następnie **New Project**.



2 W nowym oknie program poprosi o wybór początkowej aktywności, co ma przełożenie na schemat okna, od jakiego można zacząć prace nad projektem. Na początek najlepiej wybrać opcję **Empty Activity**, w której okno początkowe będzie puste. Dzięki temu łatwiej nam będzie odnaleźć się w projekcie i samodzielnie budując kolejne jego elementy, poznawać działanie programu. Wybór należy zatwierdzić przyciskiem **Next**, co spowoduje przejście do kolejnego kroku tworzenia nowego projektu.

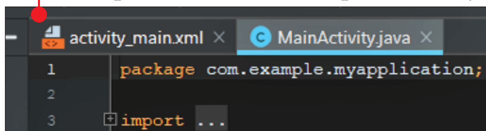


3 Oprócz określenia nazwy projektu można teraz między innymi wybrać minimalną wersję systemu Android, z którą zgodny ma być tworzony projekt. Część możliwości nie jest dostępna dla starszych wersji systemu. Z reguły powinno się wybierać najniższą wersję systemu pozwalającą na stworzenie aplikacji zgodnej z założeniami. Dlatego trzeba wiedzieć, jakie możliwości mają kolejne wersje Androida. By to sprawdzić, można kliknąć na przycisk **Help me choose**.

4 Program wyświetli zestawienie wersji Androida wraz z dodanymi do nich możliwościami. Klikając na przycisk **OK**, zamkniemy nowe okno, a aktualnie wyświetlana wersja systemu trafi do pozycji **Minimum API level** **A**. Wybór ten można jednak jeszcze zmienić, wybierając inną wersję z rozwijanej listy. Zakładamy, że w przypadku naszego pierwszego projektu nie potrzebujemy zaawansowanej funkcjonalności, dlatego możemy wybrać najstarszą wersję systemu.

5 Całość zatwierdzamy przyciskiem **Finish** **B**, co prowadzi do utworzenia projektu. Może to chwilę potrwać, jednak po pewnym czasie pojawi się okno ze skryptem **MainActivity.java**.

6 Jednocześnie powinna też zostać otwarta druga zakładka z plikiem **activity_main.xml**. Jest to plik formatu XML. Nazwa tego formatu to skrót od Extensible Markup Language, co można przetłumaczyć jako Rozszerzalny Język Znaczników. XML to uniwersalny język znaczników, który przeznaczony jest do reprezentowania różnych danych w strukturalizowany sposób. W przypadku tego pliku reprezentuje on wygląd aktywności.



Tworząc aktywności, należy tak naprawdę utworzyć dwa pliki – jeden napisany w XML z opisem wyglądu i drugi w Javie z opisem działania tej aktywności.

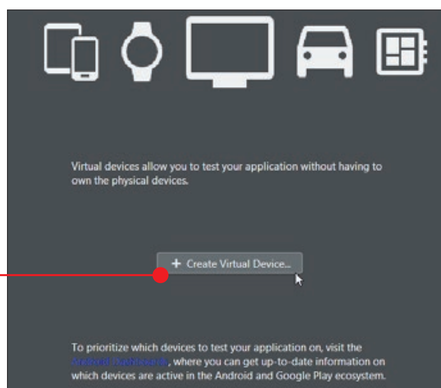
Emulator z maszyną wirtualną

Przed rozpoczęciem właściwej pracy nad projektem warto zadbać o środowisko testowe dla tworzonego programu. Testy można prowadzić zarówno na rzeczywistym urządzeniu podłączonym poprzez kabel USB, jak i na emulatorze. Emulator jest częścią **Android SDK**, ale trzeba jeszcze mieć sam system Android w odpowiedniej wersji, by móc uruchomić go w emulatorze.

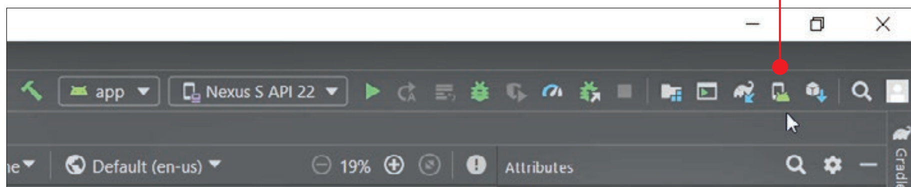
1 Klikamy na przycisk **AVD Manager** znajdujący się na pasku po prawej stronie, u góry okna programu. Otwiera on okno menedżera wirtualnych urządzeń.

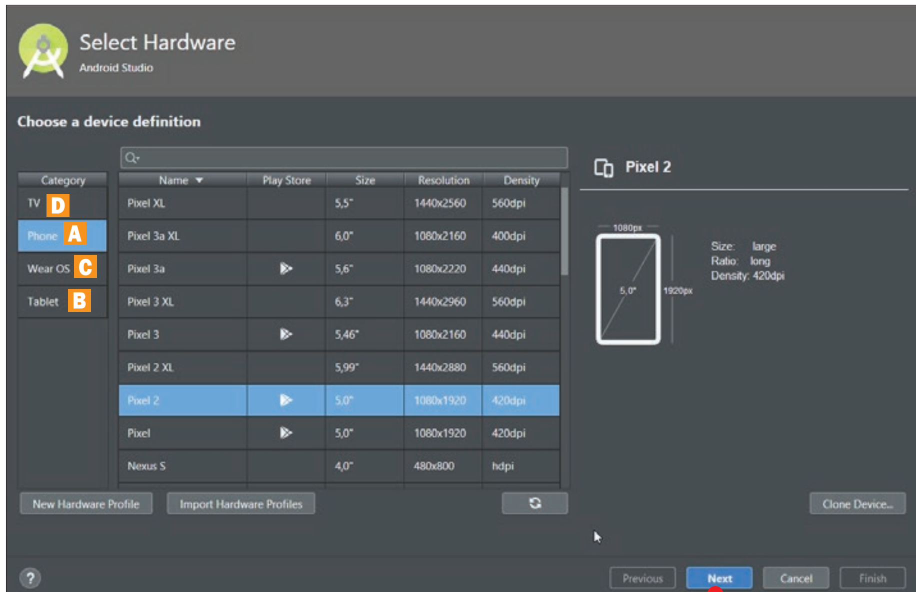
2 W nowym oknie klikamy na przycisk **Create Virtual Device** – by przejść do tworzenia wirtualnego urządzenia.

3 Program pozwala na wybranie modelu urządzenia, jakie ma być emulowane. W oknie mamy do wyboru nie tylko smartfony **A** i tablety **B**, ale także inteligentne ubrania **C** czy telewizory **D**. Takie urządzenia

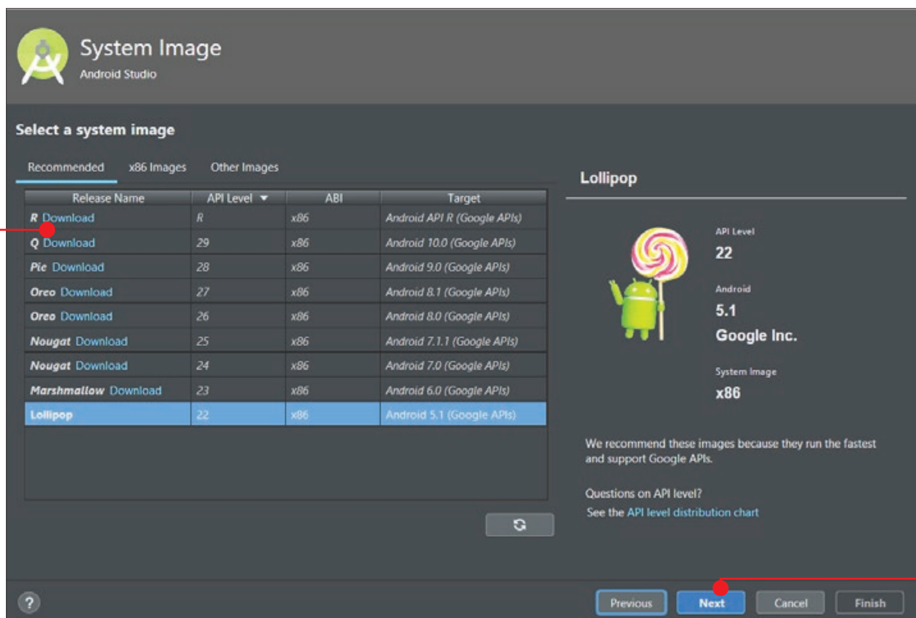


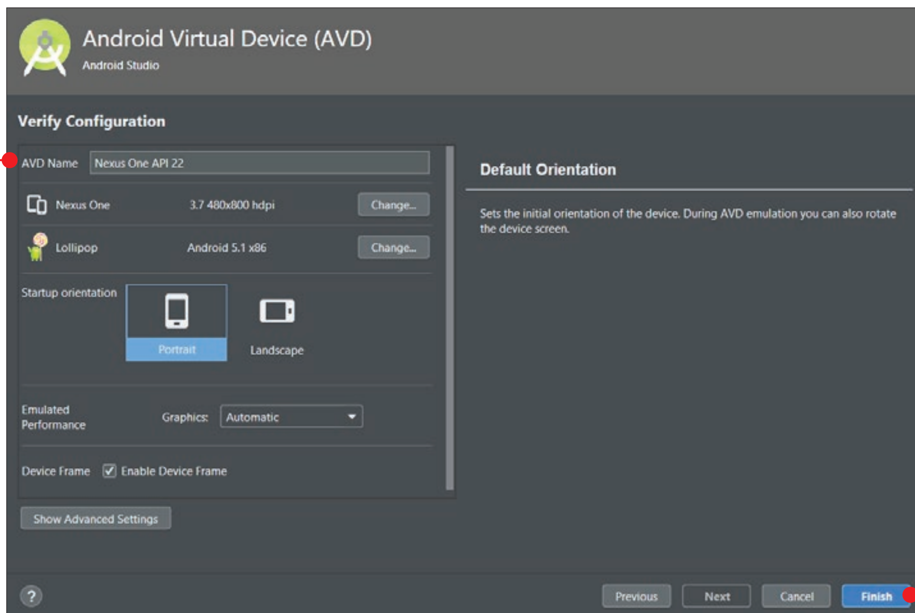
też można emulować, bo przecież i na nie można tworzyć aplikacje poprzez Android Studio. W naszym przykładzie wybieramy jeden z modeli smartfonów, którego pracę będziemy emulować na komputerze. Modele te różnią się głównie rozdzielczością i wielkością ekranu. Po dokonaniu wyboru jednego ze smartfonów klikamy na przycisk **Next**, aby móc potwierdzić wybór i przejść do kolejnego kroku.





4 Następnie należy wybrać wersję systemu do emulowania, a potem ją pobrać. Przy wersjach Androida widocznych na liście są przyciski **Download** – klikamy na jeden z nich. Nie ma dużego znaczenia, którą z rekomendowanych wersji pobierzemy. Gdy zostanie ona już pobrana, zaznaczamy ją i klikamy na **Next**, by przejść dalej.





5 W kolejnym kroku możemy między innymi nadać nazwę swojemu wirtualnemu urządzeniu, ale też zmienić decyzję dotyczącą modelu i wersji systemu, które zostały wybrane w poprzednich krokach. By wszystko zatwierdzić, klikamy na przycisk **Finish**.

Program na Androida: generator liczb

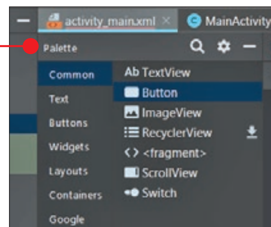
By zapoznać się z programowaniem na Androida, stworzymy prostą aplikację generującą sześć unikalnych liczb. Taką aplikację można zastosować do wytypowania liczb w popularnej loterii, w której należy postawić na sześć liczb z zakresu 1–49 wylosowanych przez maszynę losującą.

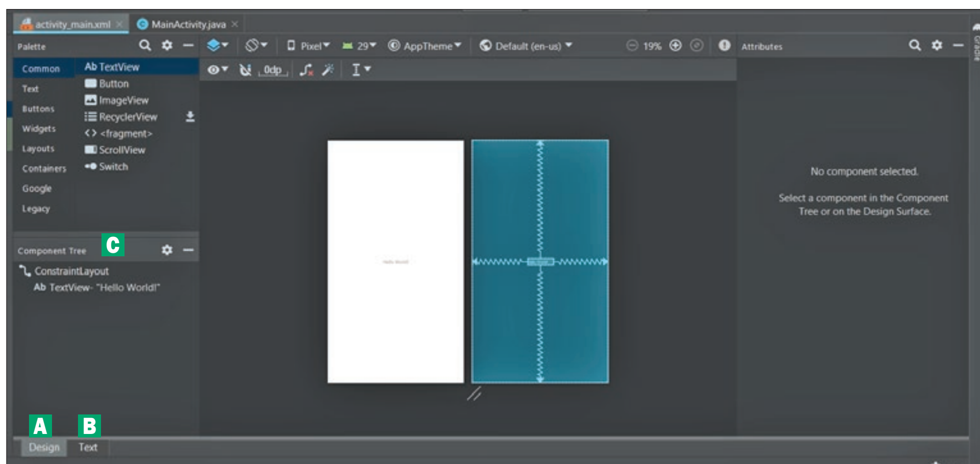
Budowa interfejsu programu

Pracę nad projektem można rozpocząć od pliku XML. Można go edytować na dwa sposoby – graficznie i tekstowo. Domyślnie otwarta zostaje edycja graficzna. Pomiędzy sposobami edycji można przełączać się przyciskami **Design** **A** i **Text** **B** u dołu sekcji po lewej stronie.

1 Domyślnie w tej aktywności znajduje się na ekranie komponent **TextView** z napisem **Hello World**, który można usunąć. Wystarczy zaznaczyć element w sekcji **Component Tree** **C** i nacisnąć klawisz delete.

2 Zamiast usuniętego właśnie elementu można dodać przycisk. Z sekcji **Palette** przeciągamy **Button** na podgląd aktywności.





3 Co ważne, miejsce, w którym widać przycisk, jest tylko poglądowe. Po uruchomieniu programu, bez wprowadzenia drobnych zmian w pliku XML, przycisk powędrowałby w lewy górny róg ekranu.

4 Przełączamy się zatem do trybu tekstowego, klikając na przycisk **Text** **B**.

5 Teraz dla znacznika odnoszącego się do komponentu **Button** możemy zmodyfikować **android:layout_height** i **android:layout_width**, zmieniając ich wartość z **wrap_content** ● na **fill_parent** ●, dzięki czemu przycisk zostanie rozciągnięty na całe okno programu.

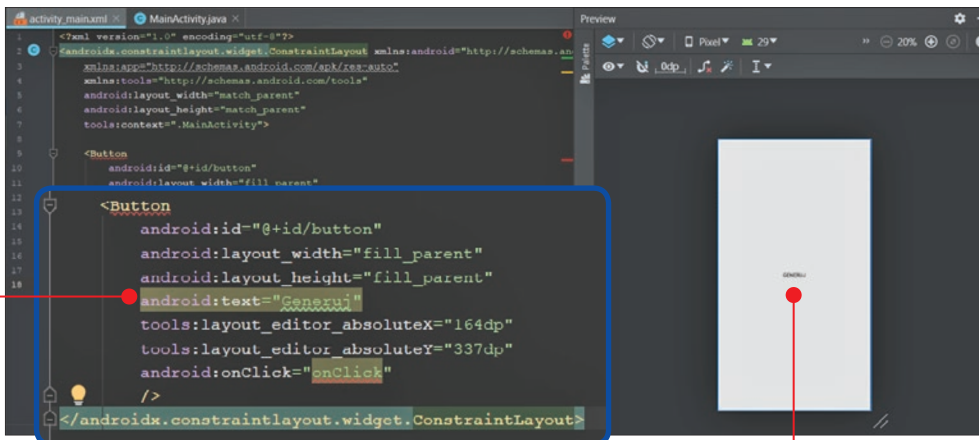
```
<Button
    android:id="@+id/button"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="Button"
    tools:layout_editor_absoluteX="164dp"
    tools:layout_editor_absoluteY="337dp"
    android:onClick="onClick"
    />
```

6 Będąc już w pliku XML, należałoby dopisać do niego, by naciśnięcie przycisku skutkowało wywołaniem metody. Można to osiągnąć, dopisując do pliku XML w sekcji odpowiedzialnej za przycisk ● – **android:onClick="onClick"** ●, gdzie w cudzysłowie podana jest nazwa metody wywoływanej zdarzeniem naciśnięcia przycisku.

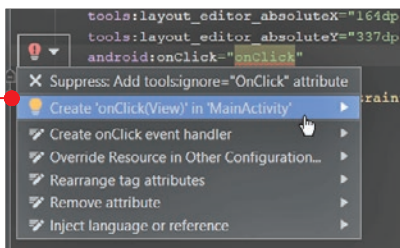
```
_main.xml ×   MainActivity.java ×
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://sc
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        tools:layout_editor_absoluteX="164dp"
        tools:layout_editor_absoluteY="337dp"
        />
</androidx.constraintlayout.widget.ConstraintLayout>
```

```
<Button
    android:id="@+id/button"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
```



7 Przycisk ten oprócz tego, że został rozciągnięty na cały ekran, pozostałe ustawienia ma domyślne, łącznie z napisem. W pliku XML zmieniamy **android:text="Button"** na **android:text="Generuj"** ●, co sprawi, że na przycisku pojawi się napis **Generuj**. Wszystkie zmiany wprowadzane w pliku XML są na bieżąco widoczne na podglądzie ●.

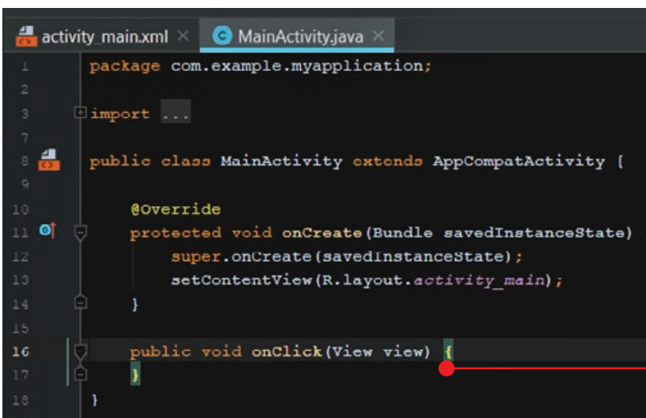


8 Metoda **onClick** musi być jeszcze zdefiniowana w pliku z Javą. Program powinien identyfikować brak definicji metody i podkreśli błąd. Można go łatwo naprawić, klikając na ikonę błędu i wybierając z rozwiniętej listy pozycję **Create 'onClick(View)' in 'MainActivity'** ●.

9 Program przełączy się automatycznie do pliku Javy, w którym znajdzie się już wygenerowana metoda **onClick** ● – cokolwiek w niej napiszemy, wykona się po naciśnięciu przycisku.

Definicja metody onClick

Przed przystąpieniem do pisania powinniśmy zrozumieć algorytm działania metody. Należy losować sześć liczb z puli 1–49. Nie mogą to być tylko liczby (pseudo)losowe z tego zakresu, ponieważ wartości wybranych sześciu liczb nie mogą się powtórzyć. Wygenerowane liczby spełniające założenia zadania będą umieszczane na liście. To kolekcja podobna do tablicy, jednak deklarując ją, nie trzeba określać jej rozmiaru. Dopóki na liście nie znajdzie się sześć liczb, będzie



generowana kolejna liczba. Jeśli wygenerowanej liczby nie ma na liście, zostanie do niej dodana.

1 Zapis tego algorytmu należy rozpocząć od deklaracji listy. Można to zrobić linijką: **List<Integer> lista = new ArrayList<>()**. Tworzy to listę o nazwie **lista**, która może zawierać wartości typu **Integer**, czyli liczby całkowite.

```
public void onClick(View view) {
    List<Integer> lista = new ArrayList<>();
}
```

2 By **List** i **ArrayList** były w ogóle rozpoznawane przez program, należy dokonać importów – odpowiednio **import java.util.List;** i **import java.util.ArrayList;**.

```
import android.os.Bundle;
import android.view.View;

import java.util.ArrayList;
import java.util.List;
```

3 Potem należy utworzyć generator liczb, czyli obiekt klasy **Random**, co można zrobić, pisząc **Random r = new Random();**.

```
public void onClick(View view) {
    List<Integer> lista = new ArrayList<>();
    Random r = new Random();
}
```

4 By klasa ta była rozpoznawana przez program, należy jeszcze dokonać importu – **import java.util.Random;**.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
```

5 Kolejny krok to użycie pętli, która będzie powtarzała operację, dopóki na liście nie znajdzie się sześć elementów. Pętlą pozwalającą na takie określenie momentu zakończenia jest pętla **while**. Tworząc ją, należy w nawiasie zapisać warunek, który musi być spełniony, aby pętla się wykonywała. Warunkiem tym jest, by rozmiar listy był mniejszy niż sześć. Rozmiar możemy sprawdzić, korzystając z polecenia **lista.size()**. Pętla powinna być utworzona w formie: **while (lista.size() < 6){ }**.

```
public void onClick(View view) {
    List<Integer> lista = new ArrayList<>();
    Random r = new Random();
    while (lista.size() < 6){

    }
}
```

6 Wewnątrz pętli trzeba zadeklarować zmienną typu **Integer** (liczbę całkowitą). Wartość tej zmiennej powinna być generowana przez obiekt klasy **Random**. Ponieważ metoda **nextInt** pozwala na określenie jedynie górnej granicy i generuje liczby od zera, trzeba skorzystać z pewnego zabiegu matematycznego. Potrzebna jest liczba od 1 do 49. Dlatego do liczby 1 należy dodać wygenerowaną liczbę mniejszą od 49 (wtedy mogą się generować liczby od 0 do 48). To spowoduje, że zmienna dostanie wartość z odpowiedniego zakresu. Zatem należy zapisać: **Integer nowe = 1 + r.nextInt(49);**.

```
public void onClick(View view) {
    List<Integer> lista = new ArrayList<>();
    Random r = new Random();
    while (lista.size() < 6){
        Integer nowe = 1 + r.nextInt( bound: 49);
    }
}
```

7 Kolejny krok polega na sprawdzeniu, czy wygenerowana liczba znajduje się już na liście. Taką operację trzeba podzielić na kilka części i zrealizować skryptem, w którym:

A **boolean czyjest = false;** – jest deklaracją zmiennej typu boolowskiego, mogącej przyjmować wartości prawda i fałsz, zmienną należy wykorzystać do przechowania informacji o tym, czy któraś z liczb na liście jest równa tej wygenerowanej,

B **for (int i:lista)** – tworzy pętlę for, która wykona się dla każdego elementu listy; w każdym kolejnym przejściu pętli zmienna **i** będzie otrzymywać wartość kolejnych elementów listy,

```
public void onClick(View view) {
    List <Integer> lista = new ArrayList <>();
    Random r = new Random();
    while (lista.size() < 6){
        Integer nowe = 1 + r.nextInt( bound: 49);
A       boolean czyjest = false;
B       for (int i:lista)
        {
C           if (i==nowe){
D               czyjest = true;
            }
        }
E       if (czyjest == false) lista.add(nowe);
    }
```

C **if (i==nowe){** – to instrukcja warunkowa porównująca element listy z wygenerowaną liczbą,
D **czyjest = true;** – gdy warunek jest spełniony, zmienna **czyjest** dostanie wartość **true**,
E **if (czyjest == false) lista.add(nowe);** – po przejściu całej pętli **for** zmienna **czyjest** będzie miała wartość **false**, jeśli ani razu nie wystąpiła na liście nowa liczba, lub wartość **true**, gdy którykolwiek z elementów listy miał taką wartość. Dlatego gdy ma ona wartość **false**, to poleceniem **lista.add(nowe)** należy dodać wygenerowaną liczbę do listy.

```
        }
        if (czyjest == false) lista.add(nowe);
    }
    String napis = "Wylosowano: ";
```

8 Następnie należałoby z wygenerowanej listy liczb ułożyć napis i wyświetlić go w oknie programu. Poleceniem **String napis = "Wylosowano: „;** tworzymy zmienną **napis**, w której przechowywany będzie ciąg znaków, jaki wyświetli się użytkownikowi. Ciąg ten powinien zaczynać się od informacji, że są to wylosowane liczby, a dalej powinny być te liczby pokazane.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    przycisk = findViewById(R.id.button);
}
```

9 Dlatego w kolejnej linii trzeba skorzystać z pętli **for** – w formie **for (int i:lista) napis += i + " ";**, co pozwoli odnieść się do każdego elementu listy i jego wartość dopisywać do zmiennej **napis**, rozdzielając kolejne liczby znakiem spacji.

```
        if (czyjest == false) lista.add(nowe);
    }
    String napis = "Wylosowano: ";
    for (int i:lista) napis += i + " ";
}
```

10 Dalej należałoby skonstruowany w ten sposób napis wyświetlić. Do tego celu można wykorzystać przycisk i zastąpić na nim napis **Generuj** innym. Aby móc umieścić coś na przycisku, należy się do niego odnieść z poziomu skryptu. By było to możliwe, przycisk musi być polem klasy **MainActivity**. Trzeba zatem napisać: **Button przycisk;** jako deklarację pola klasy.

```
public class MainActivity extends AppCompatActivity {

    Button przycisk;
```

11 By w skrypcie klasa **Button** była rozpoznawana jako element aktywności, trzeba dokonać importu – **import android.widget.Button;**.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import android.widget.Button;
```

12 Do utworzonego pola klasy trzeba przypisać przycisk znajdujący się w aktywności. To należy zrobić w metodzie **onCreate**, dopisując w niej: **przycisk = findViewById(R.id.button);** – gdzie słowo **button** jest **id** elementu z aktywności, jeśli nazwa przycisku nie była zmieniana w aktywności, a o tym mówi linijka **android:id="@+id/button"** w pliku XML.

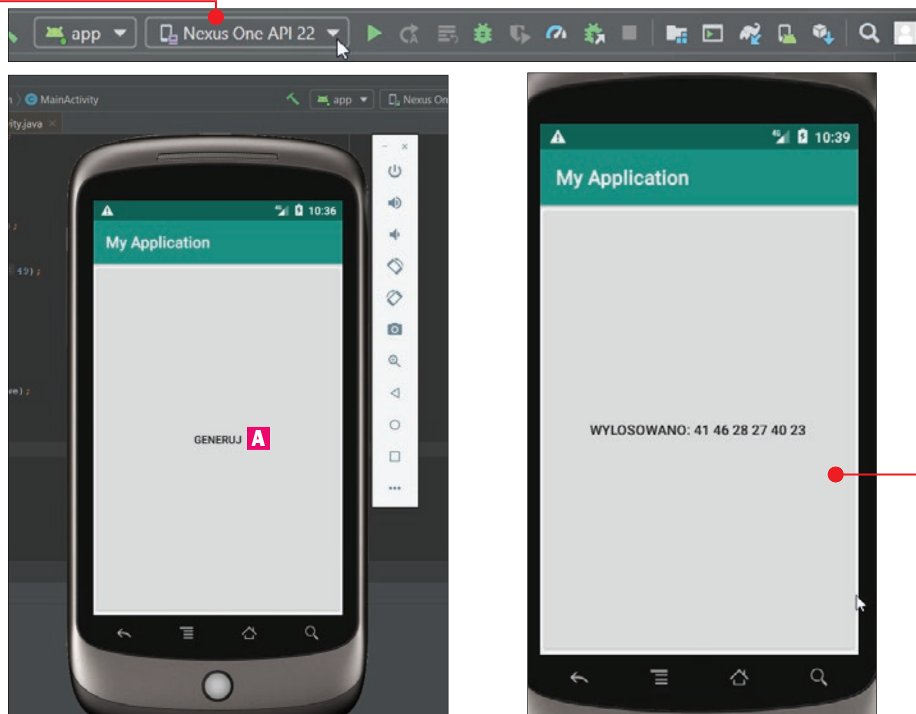
13 Na koniec metody **onClick** powinno znaleźć się po-

lecenie umieszczające napis na przycisku, czyli **przycisk.setText(napis);** ●.

```
String napis = "Wylosowano: ";
for (int i:lista) napis += i + " ";
przycisk.setText(napis);
}
```

Uruchomienie aplikacji

Sprawdźmy, czy utworzone wcześniej wirtualne urządzenie jest wybrane jako to, na którym będzie testowana aplikacja, ustawia się to w menu głównym ●. Następnie klikamy na **Run**, tuż obok. Aplikacja uruchomi się na wirtualnym urządzeniu na ekranie komputera. Kliknięcie na przycisk z napisem **Generuj** **A** powinno skutkować zmianą napisu na tym przycisku i pokazaniem zestawu wygenerowanych sześciu liczb ●.



Program na Androida: zapamiętaj kolejność

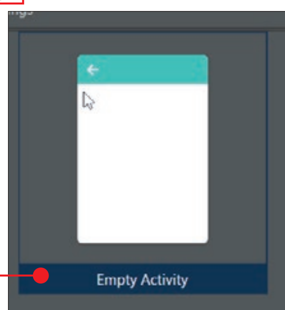
Kolejnym projektem do wykonania w Android Studio będzie gra polegająca na zapamiętaniu kombinacji podświetlanych pól i powtórzeniu jej przez naciskanie odpowiednich przycisków. Jeśli są to uda, gracz wygrywa, jeśli nie – przegrywa.

Budowanie wyglądu aplikacji

W naszej grze znajdzie się pięć przycisków. Powinny one być rozmieszczone w oknie w taki sposób, by wypełniły całą jego szerokość, a także całą wysokość. Jeśli przyciski będą ułożone w pionie, jeden pod drugim,

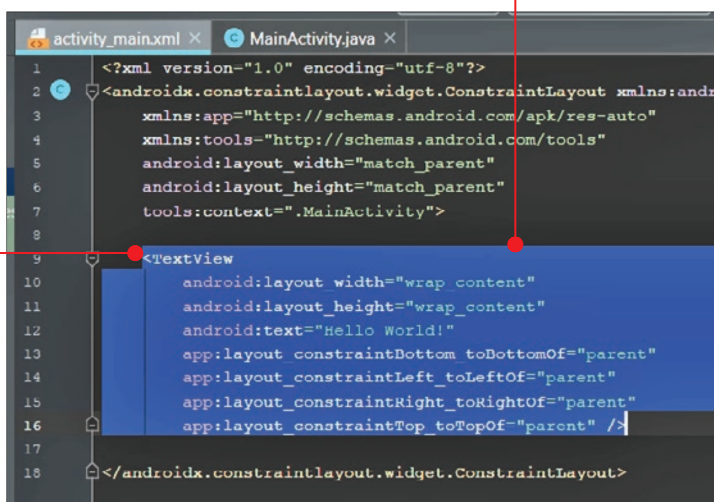
to wysokość każdego z przycisków powinna być taka sama. Tym razem wygląd aplikacji stworzymy w całości w trybie tekstowym, który w przypadku aplikacji na Androida jest chętnie wykorzystywany przez programistów.

1 Tworzymy nowy projekt w Android Studio – tak jak w poprzedniej wskazówce wybieramy jako początkowy schemat **Empty Activity** ●.

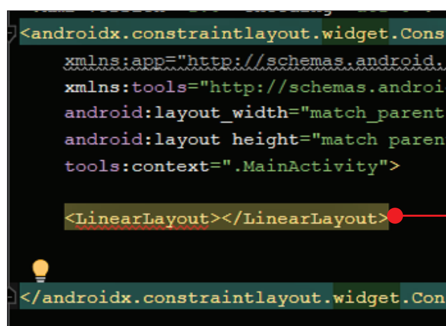


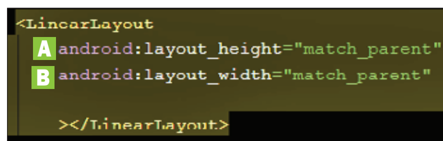
2 W otwartym pliku **activity_main.xml** przełączamy się do trybu tekstowego. Z jego poziomu usuwamy element **TextView** ●, kasując zaznaczony tekst ●.

3 Następnie należy dodać do pliku znaczniki, które spowodują utworzenie pięciu przycisków. Ponieważ przyciski będą różnić się tylko nieznacznie, można utworzyć jeden przycisk, a potem skopiować go i powielić, zmieniając tylko szczegóły, które mają różnić przyciski. Jednak by przyciski były ładnie, równomiernie ułożone w pionie, niezbędne będzie dodanie jeszcze jednego elementu – **LinearLayout**.



4 Należy utworzyć znacznik LinearLayout wraz z jego zamknięciem, czyli: **<LinearLayout> </LinearLayout>** ●.





5 W znaczniku otwierającym należy teraz dodać właściwości tego elementu okna. Powinien on swoim rozmiarem wypełnić ekran. Dlatego należy dopisać **android:layout_height="match_parent"** **A**, by wysokość elementu wypełniała ekran, i **android:layout_width="match_parent"** **B**, aby wypełnić szerokość.

6 W tym samym znaczniku musimy zapisać, aby elementy

okna, które będą umieszczane wewnątrz tego znacznika (czyli przyciski), były układane jeden pod drugim. Trzeba dodać linijkę: **android:orientation="vertical"**.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
```

7 Teraz przyszedł czas na dodanie przycisków. Ponieważ będą podobne, powielimy jeden tak, by stworzyć ich pięć. Dodajemy znacznik **Button** między znacznikami **LinearLayout**.

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <Button/>
```

8 W znaczniku **Button** trzeba ustawić id przycisku – każdy z nich będzie miał inne. Pierwszy będzie miał id **button1**, co zapisujemy **android:id="@+id/button1"**.

```
<Button
    android:id="@+id/button1"
```

9 Dopisujemy także: **android:layout_width="match_parent"**, by szerokość przycisku wypełniła cały wolny obszar.

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
```

10 By przyciski podzieliły się po równo wysokością ekranu, trzeba w nich ustawić właściwości **android:layout_height="wrap_content"** i **android:layout_weight="1"** (co ustawia „wagę" przycisku – jeśli mają one taką samą „wagę", po równo dzielą się wysokością).

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
```

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text=" "
```

11 Można też ustawić tekst na przyciskach – tak by go nie było, poleceniem **android:text=" "**.

12 Dodajemy linijkę **android:onClick="onClickP1"**, co sprawi, że do naciśnięcia przycisku przypisana zostanie metoda **onClickP1**, którą trzeba będzie zdefiniować w **MainActivity.java** (patrz kolejna strona).

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text=" "
    android:onClick="onClickP1"/>
```

13 Tak stworzony przycisk można powielić – zmieniając id (**button2**, **button3** itd.) i nazwę metody, która ma być wywoływana po naciśnięciu przycisku – **onClickP2**, **onClickP3** itd.

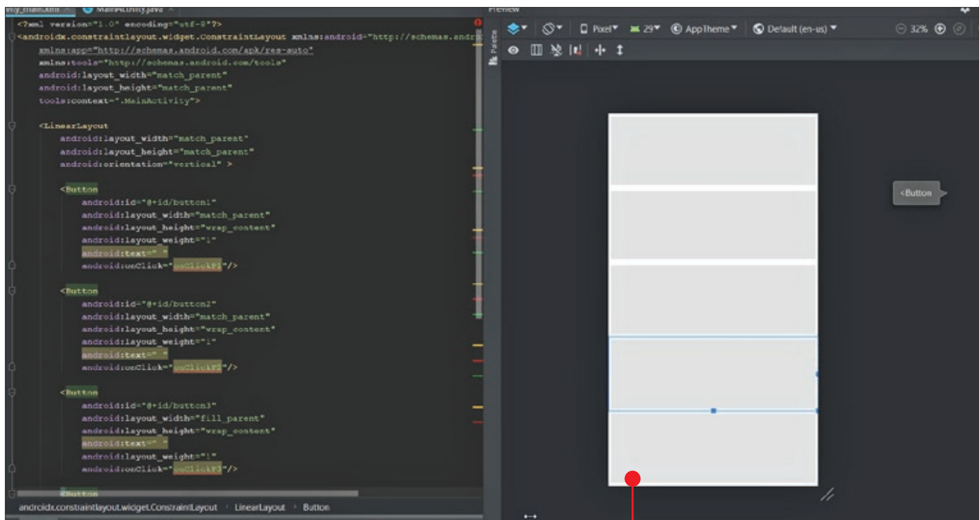
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:app="http://schemas.android.com/apk/res-
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >

        <Button
            android:id="@+id/button1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text=" "
            android:onClick="onClickP1"/>

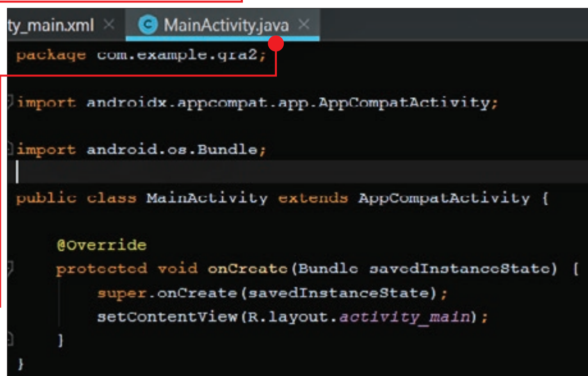
        <Button
            android:id="@+id/button2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text=" "
            android:onClick="onClickP2"/>

        <Button
            android:id="@+id/button3"
```



14 Zmiany wprowadzane w pliku XML powinny być na bieżąco widoczne na podglądzie obok. Po powieleniu przycisku podgląd powinien pokazywać pięć takich samych przycisków ułożonych jeden pod drugim.

Dalsze prace będziemy prowadzić już w pliku **MainActivity.java**.



Odniesienie do przycisków z poziomu skryptu

1 By z poziomu skryptu móc odnosić się do przycisków, powinny one być zadeklarowane jako pola klasy. Taką deklarację możemy napisać w jednej linii kodu. Jeśli deklarujemy kilka elementów tego samego typu, można wypisywać je po przecinku. Piszemy zatem: **Button p1,p2,p3,p4,p5;**.

```
Button p1,p2,p3,p4,p5;
@Override
protected void onCreate(Bund
```

2 Jednak by klasa **Button** była rozpoznana przez skrypt, trzeba dokonać jej importu, pisząc: **import android.widget.Button;**.

```
import android.os.Bundle;
import android.widget.Button;
```

3 Do zadeklarowanych pól klasy trzeba teraz przypisać odpowiednie przyciski – w metodzie **onCreate** należy napisać **p1 = findViewById(R.id.button1);**, aby przycisk o id **button1** z poziomu skryptu był przechowywany pod nazwą **p1**.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    p1 = findViewById(R.id.button1);
}
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    p1 = findViewById(R.id.button1);
    p2 = findViewById(R.id.button2);
    p3 = findViewById(R.id.button3);
    p4 = findViewById(R.id.button4);
    p5 = findViewById(R.id.button5);
}
```

4 Analogicznie trzeba postąpić z pozostałymi przyciskami.

Wygenerowanie kolejności

Przyciski będą w wygenerowanej przez program kolejności zmieniać kolor. Należy napisać skrypt, który pozwoli na generowanie tej kolejności.

1 W tym celu przyda się tablica, która będzie miała 10 pozycji. Każda kolejna pozycja będzie mogła przyjąć wartość od 1 do 5, co będzie odpowiadało numerom przycisków od p1 do p5. Tablica powinna być zadeklarowana jako pole klasy. Aby stworzyć taką tablicę, trzeba napisać **int[] kolejka = new int[10];**.

```
Button p1,p2,p3,p4,p5;
int[] kolejka = new int[10];
```

2 Następnie trzeba będzie wypełnić tablicę wartościami. Dobrze sprawdzi się tu pętla **for**, zatem w metodzie **onCreate** należy napisać **for(int i=0;i<10;i++){ }**.

```
p4 = findViewById(R.id.button4);
p5 = findViewById(R.id.button5);

for(int i=0;i<10;i++){

}
```

3 Pętla ta powinna pod kolejnymi indeksami tablicy umieszczać wygenerowane liczby. Żeby móc je generować, potrzebny jest obiekt klasy **Random**.

```
Button p1,p2,p3,p4,p5;
int[] kolejka = new int[10];
Random r = new Random();
```

4 By klasa była rozpoznana przez program, należy dodać ją do projektu, pisząc **import java.util.Random;**.

```
import android.os.Bundle;
import android.widget.Button;
import java.util.Random;
```

5 Teraz, korzystając z obiektu klasy **Random**, wewnątrz pętli **for** można wpisywać kolejne wygenerowane wartości z metody **nextInt**. Użycie tej metody z parametrem 5 spowoduje wygenerowanie takiej wartości, w której minimalna możliwa liczba to 0, a maksymalna to 4. By do tablicy trafiły liczby od 1 do 5, trzeba do tej wygenerowanej dodać jeden. Piszemy zatem: **kolejka[i] = 1 + r.nextInt(5);**.

```
for(int i=0;i<10;i++){
    kolejka[i] = 1 + r.nextInt(5);
}
```

Zapis kolorów przycisków

1 By móc cyklicznie „podświetlać” przyciski według wygenerowanej wcześniej kolejności, należy zapisać kolory, jakie mają mieć przyciski. Jeden kolor to kolor „podświetlonego” przycisku, czyli aktywnego, a drugi – nieaktywnego. Tymi kolorami będziemy posługiwać się w kilku miejscach w kodzie, najlepiej zatem zapisać je jako pola klasy, co ułatwi w przyszłości zmianę koloru. Gdy programujemy na Androida, kolory zapisujemy jako liczby całkowite. Aby uzyskać odpowiednią liczbę, potrzebna będzie metoda klasy **Color** z biblioteki **android.graphics.Color**; – trzeba więc ją importować.

```
import android.widget.Button;
import android.graphics.Color;
```

2 Następnie należy utworzyć dwa pola typu **int** – jedno o nazwie **aktywny**, a drugie **nieaktywny**.

```
Button p1,p2,p3,p4,p5;
int aktywny =
int nieaktywny =
```

```
public class MainActivity extends AppCompatActivity {

    Button p1,p2,p3,p4,p5;
    int aktywny = Color.parseColor( colorString: "#eb4034");
    int nieaktywny = Color.parseColor( colorString: "#000DC0");
```

3 Nowym polom nadaje się wartości, korzystając z metody **Color.parseColor()**, gdzie w parametrze podaje się heksadecymalny zapis koloru w systemie RGB.

Zmiana koloru przycisków

Zmiana koloru przycisków powinna odbywać się w sposób cykliczny. By wykonywać coś cyklicznie, w metodzie **onCreate** można utworzyć obiekt **CountDownTimer**. Jednak by było to możliwe, trzeba dokonać odpowiedniego importu.

```
new CountDownTimer( millisInFuture: 11000, countDownInterval: 1000) {
}.start();
```

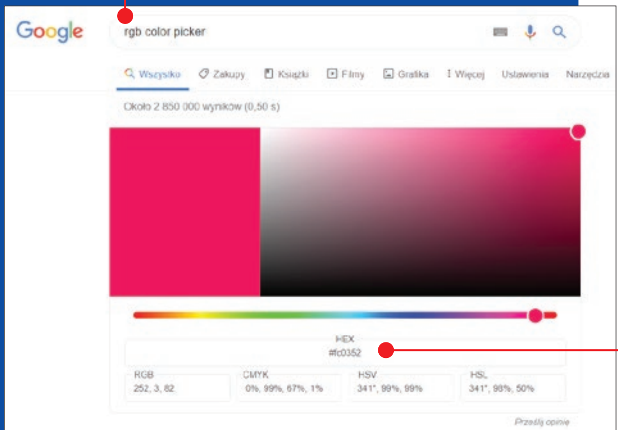
RGB

To jeden z modeli przestrzeni barw, którą opisuje się współrzędnymi **R**, **G** i **B**. Nazwa ta powstała z połączenia pierwszych liter angielskich nazw barw: **R** – red (czerwonej), **G** – green (zielonej) i **B** – blue (niebieskiej), z których model ten się składa. Jest to model wynikający z właściwości ludzkiego oka – wrażenie widzenia dowolnej barwy można wywołać przez zmieszanie w ustalonych proporcjach trzech wiązek światła o barwie czerwonej, zielonej i niebieskiej. Jest to model często stosowany do zapisu kolorów w informatyce. Ma to związek z budową urządzeń wyświetlających.

Wartość każdej z trzech składowych barwy zapisuje się liczbą z zakresu 0–255. Gdy każda z nich ma wartość zero, można powiedzieć, że „nie ma żadnej wiązki światła, w żadnym z kolorów” – w ten sposób uzyskuje się barwę czarną. Gdy wszystkie trzy składowe mają maksymalną wartość 255, wychodzi barwa biała.

Do zapisu koloru stosuje się często notację heksadecymalną (szesnastkową), system heksadecymalny – pozycyjny system liczbowy, w którym podstawą jest liczba 16. Do zapisu liczb w tym systemie potrzebne jest szesnaście znaków (cyfr szesnastkowych). System heksadecymalny to pozycyjny system liczbowy, w którym podstawą jest liczba 16. Do zapisu liczb w tym systemie potrzebne jest szesnaście znaków – cyfr. Poza tradycyjnie używanymi cyframi z systemu dziesiątkowego – od 0 do 9, wykorzystuje się też znaki A, B, C, D, E i F.

By w wygodny sposób móc dobierać kolory do programu, można w wyszukiwarce internetowej Google wyszukać frazę **rgb color picker**. Wyszukiwarka powinna wtedy wyświetlić wygodne narzędzie do wyboru barwy, dla której uzyskamy zapis RGB – także ten heksadecymalny.



```
import java.util.Random;
import android.os.CountDownTimer;
```

1 Tworząc go, podaje się dwie liczby. Pierwsza z nich to całkowity czas do odliczenia, druga – to cykliczny odstęp pomiędzy kolejnymi odliczeniami. I tak oto możemy wykonać coś zarówno po upływie całkowitego czasu określonego dla obiektu, jak i cyklicznie, co pewien określony czas.

2 To, co ma się wykonać, definiujemy w dwóch metodach nowego obiektu – odpowiednio **onFinish** (po upływie całego czasu) i **onTick** (cyklicznie).

```
new CountDownTimer( millisInFuture: 11000, countDownInterval: 1000) {

    public void onTick(long millisUntilFinished) {

    }

    public void onFinish() {

    }
}.start();
```

3 Ogólny schemat cyklicznego działania to ustawienie wszystkich przycisków na kolor nieaktywny, a następnie zmiana koloru jednego z przycisków na kolor **aktywny**.

```
public void onTick(long millisUntilFinished) {
    p1.setBackgroundColor(nieaktywny);
```

W kolorze aktywnym przyciski powinny znajdować się według wygenerowanej wcześniej kolejności. Pierwszym krokiem w metodzie **onTick** powinna być jednak zmiana koloru przycisków na **nieaktywny** – co dla pierwszego przycisku będzie miało formę polecenia: **p1.setBackgroundColor(nieaktywny);**.

4 Metodę **setBackgroundColor** trzeba wywołać też dla pozostałych czterech przycisków.

```
public void onTick(long millisUntilFinished) {
    p1.setBackgroundColor(nieaktywny);
    p2.setBackgroundColor(nieaktywny);
    p3.setBackgroundColor(nieaktywny);
    p4.setBackgroundColor(nieaktywny);
    p5.setBackgroundColor(nieaktywny);
```

5 Następnie w głównej klasie należy zadeklarować pole o nazwie **numer**, które będzie przechowywało numer aktualnie aktywnego przycisku. Pole to powinno mieć wartość początkową 0, gdyż skorzystamy z niego jako indeksu do tablicy przechowującej kolejność aktywnych przycisków, a one w tablicy **kolejka** są numerowane, począwszy od elementu o indeksie 0. Dodatkowo wartości tego pola będziemy mogli użyć do wskazania, który z kolei przycisk jest już aktywny. Z każdym kolejnym wywołaniem metody **onTick** wartość tego pola będzie rosła.

```
Random r = new Random();
int numer = 0;
```

6 W metodzie **onTick** powinno znaleźć się sprawdzanie, czy dany przycisk jest tym, który znajduje się w tablicy z kolejnością pod indeksem odpowiadającym aktualnej wartości pola **numer**. Przyciskowi, na który wskazuje kolejność, należy zmienić kolor na aktywny i napis – tak, aby wskazywał, który z kolei przycisk jest wyświetlony jako aktywny. Dla sprawdzenia, czy pierwszy przycisk (**p1**) jest tym, na który wskazuje kolejność – trzeba napisać instrukcję warunkową w formie: **if (kolejka[numer] == 1) { }**.

```
p3.setBackgroundColor(nieaktywny);
p4.setBackgroundColor(nieaktywny);
p5.setBackgroundColor(nieaktywny);

if (kolejka[numer] == 1) { }
```

```
public void onTick(long millisUntilFinished) {
    p1.setBackgroundColor(nieaktywny);
    p2.setBackgroundColor(nieaktywny);
    p3.setBackgroundColor(nieaktywny);
    p4.setBackgroundColor(nieaktywny);
    p5.setBackgroundColor(nieaktywny);


    if (kolejka[numer] == 1) {
        p1.setBackgroundColor(aktywny);
    }
}
```

7 Gdy ten warunek będzie spełniony, trzeba dla przycisku **p1** zmienić kolor tła na **aktywny**, co robi się poleceniem **p1.setBackgroundColor(aktywny);**.

8 Należy też zmienić napis na przycisku. I tu można posiłkować się wartością pola **numer**, jednak pole to uwzględnia kolejność przycisków od zera, użytkownik zaś wcale nie powinien wiedzieć o numerowaniu przycisków od zera – dla niego pierwszy aktywny przycisk powinien być oznaczony numerem **1**. Dlatego na przycisku powinna się znaleźć wartość **numer+1**. Ponieważ jest to wartość liczbowa – nie można jej bezpośrednio umieścić na przycisku, trzeba ją przerobić na tekst, co można uzyskać, korzystając z polecenia **String.valueOf()**. Całość powinna mieć zatem formę: **p1.setText(String.valueOf(numer+1));**.

```
if (kolejka[numer] == 1) {
    p1.setBackgroundColor(aktywny);
    p1.setText(String.valueOf(numer+1));
}
```

```
    if (kolejka[numer] == 1) {
        p1.setBackgroundColor(aktywny);
        p1.setText(String.valueOf(numer+1));
    }
    if (kolejka[numer] == 2) {
        p2.setBackgroundColor(aktywny);
        p2.setText(String.valueOf(numer+1));
    }
    if (kolejka[numer] == 3)  {
        p3.setBackgroundColor(aktywny);
        p3.setText(String.valueOf(numer+1));
    }
    if (kolejka[numer] == 4) {
        p4.setBackgroundColor(aktywny);
        p4.setText(String.valueOf(numer+1));
    }
    if (kolejka[numer] == 5) {
        p5.setBackgroundColor(aktywny);
        p5.setText(String.valueOf(numer+1));
    }
}
```

9 Analogiczne instrukcje warunkowe należy zbudować też dla pozostałych czterech przycisków.

10 By pole **numer** z każdym kolejnym wywołaniem metody **onTick** wskazywało na kolejny przycisk, należy zwiększać jego wartość o jeden, co można uzyskać poleceniem **numer += 1;**.

```
        p4.setText(String.valueOf(numer+1));
    }
    if (kolejka[numer] == 5) {
        p5.setBackgroundColor(aktywny);
        p5.setText(String.valueOf(numer+1));
    }

    numer += 1;
}
```

11 Na koniec opisywanej metody powinno się jeszcze „wyzerować” wartość pola **numer**, gdy osiągnie już wartość 10. W tym celu trzeba stworzyć instrukcję warunkową, która sprawdzi, czy numer ma wartość równą 10. Powinno to wyglądać następująco: **if (numer==10) { }**.

```
    p5.setText(String.valu
}

numer += 1;

if (numer == 10){
}
```

12 Gdy opisany warunek będzie spełniony, trzeba nadać polu **numer** wartość **0**, co osiągniemy, pisząc **numer = 0;**.

```
if (numer == 10){
    numer = 0;
}
```

Zakończenie odliczania

Trzeba określić jeszcze treść metody **onFinish**, która wykonuje się po odliczeniu czasu określonego przez obiekt klasy **CountDownTimer**.

1 By użytkownik wiedział, że może już zacząć klikać na przyciski w wygenerowanej przez program kolejności, można na każdym z przycisków umieścić napis **klikaj**. Dla pierwszego z przycisków (**p1**) można to zrobić poleceniem **p1.setText("klikaj");**.

```
public void onFinish() {
    p1.setText("klikaj");
}
}.start();
```

2 Analogicznie trzeba napisać to samo dla pozostałych czterech przycisków.

```
public void onFinish() {
    p1.setText("klikaj");
    p2.setText("klikaj");
    p3.setText("klikaj");
    p4.setText("klikaj");
    p5.setText("klikaj");
}
}.start();
```

3 Podobnie trzeba zrobić z kolorem przycisków – wszystkie przyciski powinny otrzymać nieaktywny kolor, co dla pierwszego przycisku można osiągnąć, pisząc **p1.setBackgroundColor(nieaktywny);**.

```
public void onFinish() {
    p1.setText("klikaj");
    p2.setText("klikaj");
    p3.setText("klikaj");
    p4.setText("klikaj");
    p5.setText("klikaj");
    p1.setBackgroundColor(nieaktywny);
}
```

```
public void onFinish() {
    p1.setText("klikaj");
    p2.setText("klikaj");
    p3.setText("klikaj");
    p4.setText("klikaj");
    p5.setText("klikaj");
    p1.setBackgroundColor(nieaktywny);
    p2.setBackgroundColor(nieaktywny);
    p3.setBackgroundColor(nieaktywny);
    p4.setBackgroundColor(nieaktywny);
    p5.setBackgroundColor(nieaktywny);
}
```

4 Analogicznie trzeba postąpić dla pozostałych czterech przycisków.

Zapisywanie kolejności kliknięć gracza

1 By zapisywać kolejność kliknięć wykonanych przez gracza, należy utworzyć pole klasy, będące tablicą liczb całkowitych (typ **int**). Do niego z każdym kliknięciem na przyciski będą trafiać numery odpowiadające tym przyciskom pod indeksami odpowiadającymi kolejnym kliknięciom w taki sposób, by ostatecznie obydwie tablice – ta z wygenerowaną kolejnością i ta z kolejnością kliknięć gracza – gdy gracz poprawnie odtworzy kolejność miały takie same wartości pod tymi samymi indeksami. Zatem, by utworzyć tablicę do przechowywania numerów odpowiadających numerom, na które gracz kliknął, piszemy **int[] wyklikane = new int[10];**.

```
int[] kolejka = new int[10];
int[] wyklikane = new int[10];
Random r = new Random();
```

2 Kiedy deklarujemy pola, należałoby jeszcze utworzyć jedno pole klasy typu boolowskiego, które będzie przechowywało informację o tym, czy gra już się rozpoczęła. A nie rozpocznie się ona, dopóki nie zakończy się sekwencja wyświetlania aktywnych przycisków i nie wykona się metoda **onFinish** z obiektu odliczającego czas. Dzięki zastosowaniu tego pola będzie można „zablokować" możliwość klikania na przyciski, gdy cała sekwencja nie zostanie odtworzona. Pole to może mieć nazwę **czyGra** – a jego deklaracja może mieć postać **boolean czyGra = false;**.

```
public class MainActivity extends AppCompatActivity {

    Button p1,p2,p3,p4,p5;
    boolean czyGra = false;
```

```
public void onClickP1(View view) {

}
```

3 Następnie należy zadeklarować metody odpowiedzialne za kliknięcia na przyciski. Mają już one określone swoje nazwy z pliku XML. Dla pierwszego z przycisków jest to **onClickP1** – co deklaruje się, pisząc **public void onClickP1(View view) { }**. By parametr klasy **View**, który jest wymagany dla metod wykonywanych przy kliknięciu na przycisk, był rozpoznany przez program, należy dokonać odpowiedniego importu. Wszystkie metody odpowiedzialne za kliknięcia na poszczególne przyciski będą bardzo podobne – różnicą będzie to, że każda odnosi się do innego przycisku. Dlatego najpierw można zdefiniować jedną z nich, a potem ją powielić.

```
import android.os.Bundle;
import android.view.View;
```

4 By „zablokować” możliwość klikania na przyciski, zanim cała ich sekwencja będzie oznaczona jako aktywna, należy skorzystać z pola **czyGra** w taki sposób, by wszystko, co ma się wykonywać w tej metodzie, było zamknięte w instrukcji warunkowej, która pozwoli wykonywać polecenia, gdy **czyGra** dostanie wartość **true**. Wystarczy do tego zapis **if (czyGra) { }**. Umieszczenie w warunku samej nazwy zmiennej typu boolowskiego sprawia, że warunek jest spełniony, gdy ma ona wartość **true**, a niespełniony – gdy **false**.

```
public void onClickP1(View view) {
    if (czyGra){
    }
}
```

5 Wartość początkowa pola **czyGra** to **false**. W żadnym miejscu skryptu nie zapisano do tej pory, by pole to miało inną wartość. Dlatego trzeba gdzieś nadać mu wartość **true**, by instrukcje z metod odpowiedzialnych za klikanie na przyciski mogły się wykonać. Najlepszym miejscem na nadanie tej wartości polu **czyGra** jest metoda **onFinish** obiektu klasy **CountDownTimer**.

```
numer += 1;

if (numer == 10){
    czyGra = true;
    numer = 0;
}
```

6 Gdy gracz kliknie na przycisk, a gra już trwa, na przycisku powinien pojawić się numer kliknięcia wykonanego przez gracza. Można ten efekt uzyskać, pisząc wewnątrz instrukcji warunkowej **p1.setText(String.valueOf(numer+1));**. Polecenie to zmienia napis na przycisku na wartość pola **numer** powiększoną o jeden. Podobnie jak wcześniej tu także stosujemy polecenie **String.valueOf()**, by wartość liczbową przerobić na tekst, bo tylko teksty można umieszczać na przycisku.

```
public void onClickP1(View view) {
    if (czyGra){
        p1.setText(String.valueOf(numer+1));
    }
}
```

7 Liczba odpowiadająca przyciskowi, na który kliknięto, powinna trafić do tablicy **wyklikane** pod indeksem odpowiadającym kliknięciu, a indeks wyznacza pole **numer**. Operację tę można zrealizować, pisząc **wyklikane[numer] = 1;** w przypadku pierwszego przycisku. W kolejnych metodach do tablicy będą trafiały inne liczby odpowiadające liczbom w nazwie metody.

```
public void onClickP1(View view) {
    if (czyGra){
        p1.setText(String.valueOf(numer+1));
        wyklikane[numer] = 1;
    }
}
```

8 Dalej należy zwiększyć wartość pola **numer** o jeden, by kolejne kliknięcie na

```
public void onClickP1(View view) {
    if (czyGra){
        p1.setText(String.valueOf(numer+1));
        wyklikane[numer] = 1;
        numer+=1;
    }
}
```

przycisk (ten lub inny) skutkowało wyświetleniem na nim następnej liczby. Ten krok realizujemy, pisząc **numer+=1;**.

To jeszcze nie jest koniec definicji metody. Jeszcze jej nie powielajmy, ponieważ trzeba do niej dodać sprawdzanie poprawności kolejności wyklikanej przez gracza.

Sprawdzanie poprawności kolejności wprowadzonej przez gracza

1 Do sprawdzenia poprawności kolejności kliknięć wykonanych przez gracza można wykorzystać dodatkową metodę. Metoda może mieć typ **boolean** i zwraca wartość **true** lub **false** – zależnie od wyniku. By metoda mogła zwracać wartość, podczas jej deklaracji należy słowo **void** zastąpić typem danych, jaki ma zwrócić – w tym wypadku będzie to **boolean**. Deklaracja metody powinna mieć więc postać: **private boolean spr(){ }**.

```
private boolean spr(){

}
```

2 Działanie metody sprawdzającej powinno polegać na porównywaniu kolejnych elementów tablicy **kolejnosc** i **wyklikane**. By odnosić się do kolejnych elementów tablic poprzez indeksy, należy wykorzystać pętlę **for**.

```
private boolean spr(){
    for(int i=0;i<10;i++){ }
}
```

3 Wewnątrz tej pętli należy poprzez instrukcję warunkową porównywać elementy tablic o tym samym indeksie – **if(wyklikane[i]!=kolejka[i]) { }**. **!=** sprawdza,

```
private boolean spr(){
    for(int i=0;i<10;i++){
        if (wyklikane[i] != kolejka[i]){ }
    }
}
```

czy wartości są różne. Jeśli są różne – warunek jest spełniony.

4 Gdy warunek jest spełniony i wartości są różne, to znaczy, że kolejność jest niepoprawna. Dlatego należy wykonać polecenie **return false;**. Przerywa to dalsze wykonywanie metody.

```
private boolean spr(){
    for(int i=0;i<10;i++){
        if (wyklikane[i] != kolejka[i]){
            return false;
        }
    }
}
```

5 Jeśli pętla się zakończy, to znaczy, że nigdy nie przerwano jej działania, czyli nigdy elementy tablic nie były różne. A to oznacza, że gracz poprawnie odtworzył kolejność. Zatem można użyć polecenia **return true;**.

```
private boolean spr(){
    for(int i=0;i<10;i++){
        if (wyklikane[i] != kolejka[i]){
            return false;
        }
    }
    return true;
}
```

6 W ten sposób wywołanie metody **spr()** będzie niosło za sobą wartość mówiącą o poprawności zadania wykonanego przez gracza. Z metody tej należy skorzystać w metodach wykonywanych po kliknięciu przycisków. Jeżeli pole **numer** dojdzie do wartości 10, należy sprawdzić, co zwraca metoda **spr()**. Można to zrobić jedną instrukcją warunkową – **if(numer==10 && spr()) { }**, gdzie mamy dwa warunki połączone spójni-

```
p1.setText(String.valueOf(numer+1));
wyklikane[numer] = 1;
numer+=1;
if (numer==10 && spr()) { }
```

kiem **i** – czyli obydwa muszą być spełnione. Ten zapis sprawia, że pole **numer** musi mieć wartość 10 oraz **spr()** musi zwracać **true**.

7 Gdy warunki są spełnione, można poinformować gracza o wygranej – na przycisku może pojawić się napis Wygrywasz, co można uzyskać, pisząc **p1.setText(„WYGRYWASZ");**.

```
p1.setText(String.valueOf(numer+1));
wyklikane[numer] = 1;
numer+=1;
if (numer==10 && spr()) {p1.setText("Wygrywasz");}
```

```
if (numer==10 && spr()) {p1.setText("Wygrywasz");}
else if (numer==10 && !spr()) {}
```

8 Należy również odpowiednio zareagować, gdy kolejność wytypowana przez gracza nie będzie prawidłowa. Do stworzonej instrukcji warunkowej trzeba dopisać sekcję **else if**, gdzie warunkiem będzie to, że pole **numer** ma wartość 10 i jednocześnie metoda **spr()** zwraca **false**. Zapisać to można tak: **else if (numer==10 && !spr())**.

9 Analogicznie do wygranej – należy poinformować gracza o przegranej,

```
activity_main.xml × | MainActivity.java ×
            wyklikane[numer] = 2;   A
            numer+=1;
            if (numer==10 && spr()) {p2.setText("WYGRYWASZ");}
            else if (numer==10 && !spr()) {p2.setText("Przegrywasz");}
        }
    }

    public void onClickP3(View view) {   B
        if (czyGra){
            p3.setText(String.valueOf(numer+1));
            wyklikane[numer] = 3;
            numer+=1;
            if (numer==10 && spr()) {p3.setText("WYGRYWASZ");}
            else if (numer==10 && !spr()) {p3.setText("Przegrywasz");}
        }
    }

    public void onClickP4(View view) {   C
        if (czyGra){
            p4.setText(String.valueOf(numer+1));
            wyklikane[numer] = 4;
            numer+=1;
            if (numer==10 && spr()) {p4.setText("WYGRYWASZ");}
            else if (numer==10 && !spr()) {p4.setText("Przegrywasz");}
        }
    }

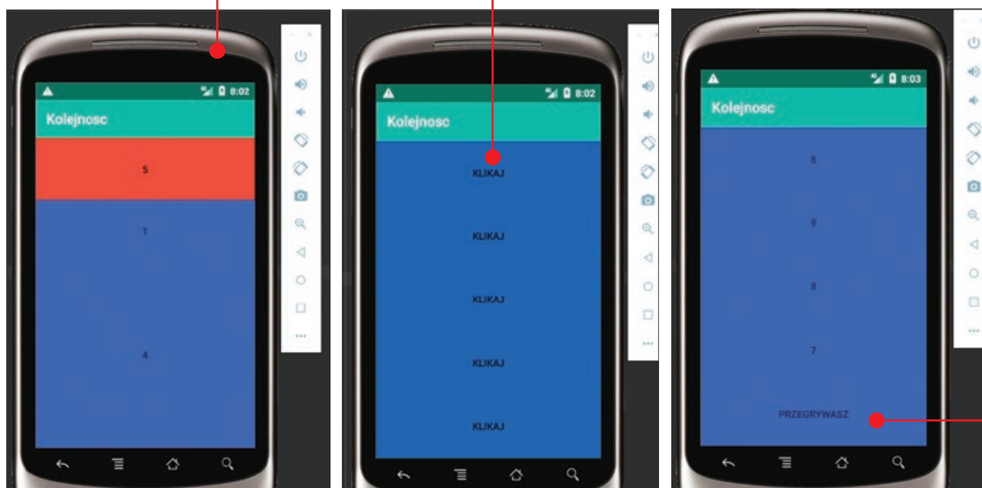
    public void onClickP5(View view) {   D
        if (czyGra){
            p5.setText(String.valueOf(numer+1));
            wyklikane[numer] = 5;
            numer+=1;
            if (numer==10 && spr()) {p5.setText("WYGRYWASZ");}
            else if (numer==10 && !spr()) {p5.setText("Przegrywasz");}
        }
    }
}
```

```
public void onClickP1(View view) {
    if (czyGra){
        p1.setText(String.valueOf(numer+1));
        wyklikane[numer] = 1;
        numer+=1;
        if (numer==10 && spr()) {p1.setText("Wygrywasz");}
        else if (numer==10 && !spr()) {p1.setText("Przegrywasz");}
    }
}
```

– co można zrobić poleceniem **p1.setText(„Przegrywasz");** ●.

10 W ten sposób kończy się definicja metody **onClickP1** – taką definicję można powielić, by powstały metody **onClickP2** **A**, **onClickP3** **B**, **onClickP4** **C** i **onClickP5** **D**, które będą odnosić się w swojej treści odpowiednio do przycisków **p2**, **p3**, **p4** i **p5**.

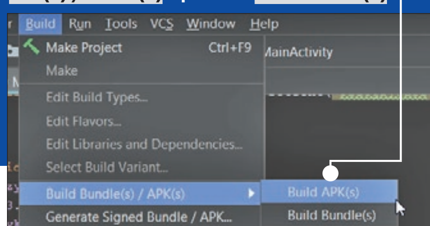
Uruchamiamy program. Przyciski w jego oknie będą w wygenerowanej kolejności zmieniać kolor ●. Gdy sekwencja się zakończy, na każdym z przycisków pojawi się napis **KLIKAJ** ●, by gracz rozpoczął wybieranie swojej sekwencji. Gdy gracz kliknie po raz dziesiąty, na ostatnio wybranym przycisku pojawi się informacja o wygranej lub przegranej ●.

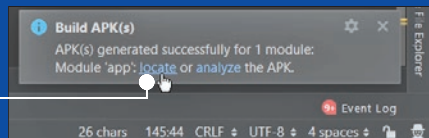


PLIK INSTALACYJNY

Aplikacje na Androida instaluje się z plików APK. Stworzony projekt można wyeksportować do takiego pliku.

1 Z menu głównego rozwijamy pozycję **Build**, następnie wybieramy **Build Bundle(s) / APK(s)** i potem **Build AKP(s)** ●.





2 Program wyświetli komunikat o wygenerowaniu pliku – pojawi się on w prawym dolnym rogu okna programu. Klikamy na **locate** ● w treści komunikatu.

3 Spowoduje to otwarcie Eksploratora z lokalizacją, w której znajduje się plik instalacyjny stworzonego projektu.

7 Java: podsumowanie

Poznaliśmy już wiele przydatnych poleceń i instrukcji z języka Java. Instrukcje te można układać w algorytmy pozwalające na rozwiązanie skomplikowanych zagadnień. Część z nich z pewnością można wykorzystać także w innych projektach. Wiele jest algorytmów uniwersalnych, które można stosować wielokrotnie

Metoda umieszczająca losowe wartości liczbowe w tablicy

W zaprezentowanym skrypcie:

A – jest deklaracją tablicy, w której będą przechowywane liczby całkowite.
B – jest deklaracją metody **dajlosowe**.
C – tworzy w metodzie obiekt klasy **Random**. Trzeba pamiętać, by umieszczać w projekcie **import java.util.Random;**, aby klasa ta mogła być rozpoznana przez program.

```
7
8 ⊟ import java.util.Random;
```

D – to pętla **for**, która pozwoli wykonać coś 10 razy.

```
A int[] zestaw = new int[10];

B private void dajlosowe(){
  C  Random r = new Random();
  D  for (int i = 0;i<10;i++){
  E    zestaw[i] = r.nextInt(10);
     }
  }
```

E – wpisuje do tablicy pod indeksem odpowiadającym aktualnej wartości zmiennej **i** (która rośnie z każdym przejściem pętli o 1) wygenerowanej wartości od 0 do 9 (10 jako granica nie może być wygenerowane).

GENEROWANIE WARTOŚCI PSEUDOLOSOWYCH

W zaprezentowanym przykładzie dla obiektu klasy **Random** wywoływana jest metoda **nextInt**. Pozwala na wygenerowanie liczby całkowitej. Jednak nie tylko liczby całkowite może generować taki obiekt. Metod tych jest znacznie więcej, a niektóre z nich to:

METODA	CO DAJE
nextBoolean()	Zwraca wartość **true** lub **false**.
nextDouble() i nextFloat()	Zwracają wartość z zakresu 0.0–1.0. Wyniki dawane przez te dwie metody różnią się jednak dokładnością (precyzją). Wynik w typie **double** oznacza możliwość przechowania go na większej liczbie bitów, zatem ułamek ten może mieć większą precyzję (co sprowadza się do większej liczby cyfr po przecinku).
nextInt()	Metody **nextInt** można użyć także bez uzupełniania jej parametru mówiącego o granicy. Wygeneruje się wtedy liczba mieszcząca się w całym zakresie typu danych **Integer**.

Metoda wyznaczająca najmniejszą liczbę z tablicy

Innym przydatnym algorytmem jest wyznaczanie minimalnej wartości z tablicy.

1 W tym celu można zadeklarować metodę **dajnajmniejsza**. By wywołanie metody niosło za sobą zwracaną wartość, metoda powinna mieć typ inny niż **void**. Załóżmy, że metoda odnosić się będzie do tablicy wypełnionej liczbami całkowitymi (może to być tablica z poprzedniej porady). Napiszmy zatem **private int dajnajmniejsza(){ }**.

```
private int dajnajmniejsza(){

}
```

2 Najmniejsza wartość z tablicy będzie przez metodę przechowywana w zmiennej. Utwórzmy zmienną o nazwie **minimum** tego samego typu co cała metoda. Jako wartość początkową należy dać jej pierwszy element tablicy, z której najmniejszą wartość ma wyznaczać metoda. Napiszmy zatem **int minimum = zestaw[0];**.

```
private int dajnajmniejsza(){
    int minimum = zestaw[0];
}
```

3 Dalej należy poprzez pętlę odnosić się do kolejnych elementów tablicy. Dlatego tworzymy pętlę **for**, która będzie zwiększała wartość wewnętrznej zmiennej o 1, począwszy od zera aż do 10, bo tyle elementów ma tablica.

```
private int dajnajmniejsza(){
    int minimum = zestaw[0];
    for(int i=0;i<10;i++){

    }
}
```

4 Wewnątrz pętli **for** należy umieścić instrukcję warunkową. Powinna ona w swoim warunku sprawdzać, czy zmienna

minimum ma wartość większą niż element tablicy, do którego można odnieść się poprzez indeks odpowiadający zmiennej wewnętrznej z pętli **for**. Powinno to być zapisane jako **if (minimum > zestaw[i]) { }**.

```
private int dajnajmniejsza(){
    int minimum = zestaw[0];
    for(int i=0;i<10;i++){
        if (minimum>zestaw[i]){
        }
    }
}
```

5 Kiedy opisany warunek będzie prawdą, należy nadać zmiennej **minimum** nową wartość, która powinna być równa **zestaw[i]**, czyli elementowi tablicy, który był

```
private int dajnajmniejsza(){
    int minimum = zestaw[0];
    for(int i=0;i<10;i++){
        if (minimum>zestaw[i]){
            minimum = zestaw[i];
        }
    }
}
```

mniejszy (lub równy) wcześniejszej wartości zmiennej **minimum**.

6 Kiedy pętla **for** zakończy swoje działanie, to znaczy, że sprawdzone zostały wszystkie elementy tablicy, a zmienna minimum jest równa temu najmniejszemu. Można napisać **return minimum;**, co sprawi, że wartość zmiennej **minimum** będzie zawsze zwracana jako wartość wywołania metody. Przedstawiony algorytm wyznaczania najmniejszej liczby z zestawu jest uniwersalny. Mając tablicę z liczbami w innym typie danych (na przykład **double** czy **float**), wystarczy zadeklarować zmienną **minimum** i samą metodę – jako ten sam typ danych – a działanie algorytmu pozostaje niezmienne.

```
private int dajnajmniejsza(){
    int minimum = zestaw[0];
    for(int i=0;i<10;i++){
        if (minimum>zestaw[i]){
            minimum = zestaw[i];
        }
    }
    return minimum;
}
```

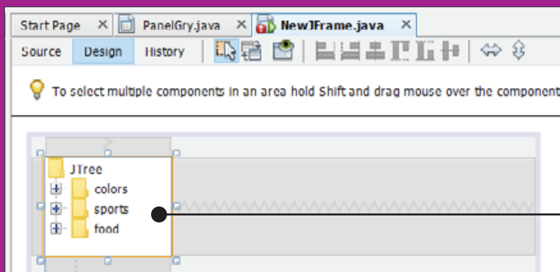
Metoda wyznaczająca największą liczbę z tablicy

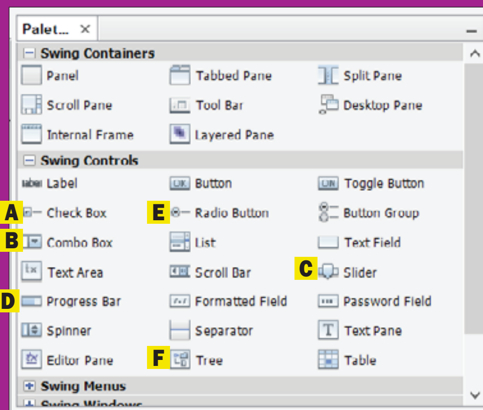
Na tej samej zasadzie co w poprzednim przykładzie można napisać metodę zwracającą największą liczbę z tablicy. Wtedy wewnątrz metody należy zadeklarować zmienną **maksimum**. Przechodząc po kolejnych elementach tablicy, trzeba sprawdzać, czy są one większe od aktualnej wartości zmiennej **maksimum**. Jeśli jakiś element tablicy ma wartość większą od tej zmiennej, zmienna ta powinna dostać nową wartość równą temu elementowi. Tak jak w wypadku najmniejszej z liczb po przejściu całej pętli wartość zmiennej **maksimum** powinna być zwrócona (**return maksimum;**) jako wynik działania metody.

```
private int dajnajwieksza(){
    int maksimum = zestaw[0];
    for(int i=0;i<10;i++){
        if (maksimum<zestaw[i]){
            maksimum = zestaw[i];
        }
    }
    return maksimum;
}
```

PALETA KONTROLEK

Budując programy z oknem graficznym, możemy korzystać z palety kontrolek. Znajduje się tam wiele ciekawych kontrolek, które mogą być przydatne w różnego rodzaju aplikacjach. Na przykład **Check Box** A wykorzystuje się do tworzenia list wielokrotnego wyboru, pole tego typu może być zaznaczone lub nie. Często można zobaczyć Check Boxy przy akceptacji regulaminów itp. Innym ciekawym elementem jest **Combo Box** B, czyli rozwijana lista elementów do wyboru.





Warto zwrócić uwagę także na **Slider** C, czyli suwak.

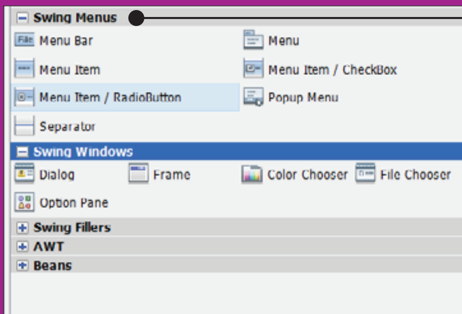
Wśród kontrolek gotowych do wykorzystania w projektach znajduje się też **Progress Bar** D. Takiej kontrolki można użyć do pokazywania postępów w pracy aplikacji.

Z kolei z kontrolki **Radio Button** E można korzystać, kiedy mamy do zaprogramowania dla użytkownika odpowiedź jednokrotnego wyboru. Kontrolki tego typu połączone w jedną grupę będą ze sobą współpracować tak, że tylko jedna z nich będzie mogła być zaznaczona. To przydatne, gdy musimy stworzyć aplikację, która będzie testować wiedzę użytkownika poprzez quiz, w którym tylko jedna odpowiedź na dane pytanie może być poprawna.

W palecie znajdują się też bardziej rozbudowane kontrolki – tu należy zwrócić uwagę na **Tree** F. Kontrolka ta pozwala na wyświetlanie drzewa lokalizacji w oknie programu.

Oprócz kontrolek widocznych w kategoriach **Swing Controls** i **Swing Containers** warto też poznać inne kategorie, jak na przykład **Swing Menus**. Tu znajdziemy kontrolki, które pozwolą na budowę menu aplikacji w pełni zgodnego z naszymi założeniami projektu.

Wśród kontrolek znajdziemy też kategorię **AWT**, a w niej kontrolki przypominające to, co widzimy w **Swing Controls**, jest to jednak starsza biblioteka zawierająca kontrolki graficzne.



8 Testy

Testowanie jest ważnym aspektem tworzenia oprogramowania. Pozwala uniknąć sytuacji, gdy do użytkowników trafia oprogramowanie z błędami. Błędy – wiadomo – mają negatywny wpływ nie tylko na funkcjonalność programu, ale też na wizerunek twórców

W wypadku oprogramowania tworzonego komercyjnie testy mają duże znaczenie dla budżetu projektu – im wcześniej wykryjemy błąd, tym niższy jest koszt jego naprawienia. Należy zatem wykryć błędy w najwcześniejszej możliwej fazie, w trakcie pisania kodu programu. Dlatego każdy szanujący się programista powinien testować kod, który napisze. Oddając kod do użytku, powinien być pewny, że działa on tak, jak powinien. Jak organizować testy? Czy należy wielokrotnie uruchamiać pisany program i sprawdzać różne możliwe do wykonania przez użytkownika operacje? Czy nie będzie to zbyt żmudne i czasochłonne? Czy taki sposób testowania byłby efektywny? Na szczęście inżynieria oprogramowania jako nauka wypracowała odpowiednie mechanizmy – testy jednostkowe.

Testy jednostkowe

Test jednostkowy jest to działanie testowe tworzonego przez nas oprogramowania. Testy te opierają się na pisaniu metod testujących krótsze części kodu, tak zwane jednostki – stąd ich nazwa.
Weryfikują one poprawność działania pojedynczych elementów, takich jak metody, obiekty lub procedury. To, co testujemy, zależy od wybranej przez nas ścieżki programowania.
Wykonując taki test, uzyskujemy wynik, który następnie należy porównać z oczekiwanym rezultatem. Będzie on albo pozytywny, albo negatywny – wpisze się w ramy naszych oczekiwań lub też nie.
Robi się to po to, by uniknąć skutków, jakie może powodować niewykryty błąd. Im szybciej wyłapiemy nieprawidłowość, tym mniejsze będzie spustoszenie w kodzie programu w dalszej części pracy.
Jeżeli chcemy poświęcić czas na testowanie, należy przygotować się na to już w trakcie pisania kodu – tak aby możliwe było wykonywanie na nim sprawdzianów.
Jednak należy pamiętać, że nie cały kod da się pokryć testami. Trzeba to zrobić w wypadku najbardziej newralgicznych fragmentów. Idealną sytuacją jest, gdy mamy testy z wymaganą funkcjonalnością i kod, który spełnia ich założenia. Uruchamiając test, dowiemy się, czy program działa prawidłowo, bez rzeczywistego uruchamiania całego programu. To niewątpliwa zaleta testów jednostkowych. Wykonywanie ich na bieżąco umożliwia wykrycie błędu i jego lokalizacji we fragmencie kodu, zanim trafi on do programu.
Wymienione powody sprawiają, że testy jednostkowe są popularne w programowaniu ekstremalnym.

Podział testów jednostkowych

Testy jednostkowe można podzielić na kilka kategorii: analiza ścieżek; użycie klas równoważności; testowanie wartości brzegowych; testowanie składniowe.

■ W teście, który wykorzystuje **analizę ścieżek**, określamy punkt początkowy oraz punkt końcowy dla przeprowadzenia testów i badamy przebieg możliwych ścieżek pomiędzy nimi. Możliwe ścieżki od punktu początkowego do punktu końcowego dzielimy na dwa podejścia:

- przetestowaniu podlega każda możliwa ścieżka w każdej funkcji,
- pętla uniemożliwia wykonanie testu na ścieżce.

PROGRAMOWANIE EKSTREMALNE

To sposób programowania mający na celu wydajne tworzenie małych i średnich projektów wysokiego ryzyka, czyli takich, w których nie wiadomo do końca, co się tak naprawdę robi i jak to prawidłowo zrobić, bo zazwyczaj nie są one opracowywane według ściśle opisanego projektu, ale projektowane na bieżąco. Jest to koncepcja prowadzenia projektu informatycznego na podstawie obserwacji innych projektów, które odniosły sukces. Jednym z elementów programowania ekstremalnego jest na przykład pisanie w parach, gdy jedna osoba tworzy kod, a druga kontroluje piszącego i sprawdza poprawność napisanego kodu. Programiści pracujący w ten sposób zamieniają się rolami na kolejnych etapach. Inną cechą programowania ekstremalnego jest stała współpraca z osobą zlecającą stworzenie oprogramowania – często bez wytworzenia specyfikacji na bieżąco wprowadza się modyfikacje na podstawie zaleceń zleceniodawcy.

W sytuacji gdy zastosowaliśmy w kodzie pętlę i niemożliwe jest wykonanie testu jednostkowego, możemy zastosować dwie grupy testów: **Boundary test** (działania w pętli nie są wykonywane lub działania w każdej pętli są wykonywane raz i dodatkowo wszystkie ścieżki wewnątrz pętli są wykonane raz, po wykonaniu testu zwracana wartość to zero lub jedno przejście) i **Interior test** (działania we wnętrzu pętli uważa się za przetestowane, jeśli zostały wykonane wszystkie ścieżki, które są możliwe przy dwukrotnym powtórzeniu pętli, po wykonaniu testu zwracana wartość to dwa przejścia).

■ Podczas przeprowadzania **testu klas równoważności** używa się klasy równoważności, które są zbiorem danych o podobnym sposobie przetwarzania. Aby klasa została uznana za poprawną, test wykonuje się jedynie na kilku elementach zbioru. Klasy równoważności dzielą się na:

- **klasy poprawności** – przewidujemy poprawne wykonanie programu,
- **klasy niepoprawności** – przewidujemy błędne wykonanie programu.

■ Kolejny rodzaj testu jednostkowego – **test wartości brzegowych** – to rozwinięcie testu klas równoważności. Wartością brzegową nazywamy element zbioru klasy równoważności, który znajduje się w pobliżu jej granicy. Wartości, które testujemy, to takie, które należą do zbioru, ale także te, które nie należą do niego, ale ich wartość jest zbliżona do granicznych wartości zbioru.

■ **Testowanie składniowe**, czyli kolejny rodzaj testu jednostkowego, opiera się na sprawdzeniu poprawności danych wprowadzanych do systemu. Możliwe błędy są zależne zarówno od systemu, jak i środowiska:

- wymuszone wartości pól (bazy danych),
- autokorekty (MS Office).

Zasada „garbage in – garbage out” jest podstawą w testowaniu tego rodzaju, jednak zadziała, jedynie gdy brakuje mechanizmu sprawdzania danych na wejściu lub testów na tolerancję systemu na błędne dane.

JUnit

Do tworzenia powtarzalnych testów jednostkowych oprogramowania napisanego w języku Java służy narzędzie **JUnit**. NetBeans IDE wykorzystywane do programowania w poprzednich rozdziałach ma wbudowane mechanizmy tworzenia plików testowych dla narzędzia JUnit. Skorzystamy z nich, jednak najpierw należy utworzyć pusty projekt, do którego można dodać plik z klasą. W pliku tym możemy umieścić następujący skrypt ●. To prosta klasa zawierająca metodę dotyczącą listy liczb całkowitych i typu boolowskiego, pozwalającą sprawdzić, czy podana liczba znajduje się na liście. W ramach ćwiczenia możemy przeprowadzić testy sprawdzające, co zwraca metoda, gdy podamy jej konkretne liczby do sprawdzenia.

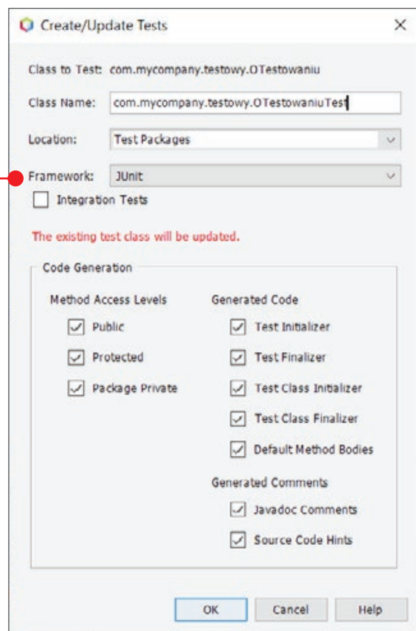
```
package com.mycompany.testowy;
/**...4 lines */
public class OTestowaniu {
    public int[] liczby = new int[10];
    public static void main(String args[]) {

    }
     public OTestowaniu() {
        for (int i = 0;i<10;i++)
        {
            liczby[i] = i*(i+1);
        }
    }

     public boolean czyZawiera(int x){
         for(int i:liczby){
             if (x==i) return true;
         }
         return false;
     }
}
```

1 Najpierw trzeba wygenerować plik z testami. W tym celu rozwijamy opcję **Tools** z menu górnego i wybieramy pozycję **Create/Update tests** ●.

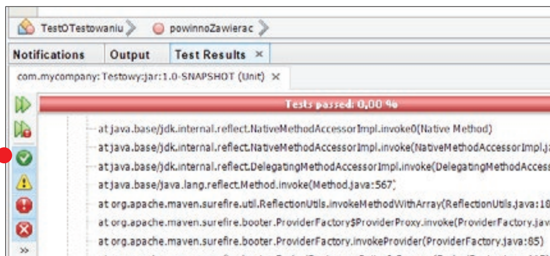
2 W nowym oknie możemy wybrać lokalizację pliku w projekcie i **Framework** ●.



3 Program wygeneruje oddzielny plik z testami jednostkowymi. Każda metoda z klasy ma w pliku z testami stworzony swój odpowiednik do testów. I tak dla metody typu boolowskiego z klasy głównej w pliku z zapisem testów znajduje się metoda **testCzyZawiera** ●, której definicja może już zawierać pewien rodzaj testu jednostkowego.

```
@Test
public void testCzyZawiera() {
    System.out.println("czyZawiera");
    int x = 0;
    OTestowaniu instance = new OTestowaniu();
    boolean expResult = false;
    boolean result = instance.czyZawiera(x);
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    fail("The test case is a prototype.");
}
```

4 Działanie tej metody polega na określeniu, czy wartość zwracana przez metodę **czyZawiera** jest taka sama jak wartość oczekiwana dla danej liczby. Oczekiwany wynik zwrotu dla konkretnej liczby może przyjąć wartość **true** lub **false**. Po uruchomieniu testów program poinformuje nas o ich wynikach ●. Wskaże jaki procent z nich został zaliczony. Jeśli testy mają wynik **failure** – oznacza to, że oczekiwany wynik zwracany przez metodę był inny niż ten, jaki faktycznie został zwrócony. W takim wypadku trzeba przeanalizować definicję metody – być może popełniono w niej błąd logiczny podczas tworzenia algorytmu jej działania.



9 Logi

Nawet najlepiej przetestowane oprogramowanie może nie uchronić się przed występowaniem drobnych niepoprawności. W celu ich wyłapania często stosuje się zapis wykonywanych przez użytkownika czynności

Pliki zawierające chronologiczny zapis zdarzeń – czynności wykonywanych przez użytkownika programu – są nazywane logami. Pliki te są tworzone przez programy automatycznie, jednak to programiści muszą stworzyć system zapisywania logów z programów.

Po co stosować logi

Logi są używane nie tylko do wykrywania ewentualnych nieprawidłowości w działaniu programu, ale też do analizowania pracy systemu informatycznego, na przykład sporządzania statystyk czy wykrywania prób włamań do systemu wraz ze sposobem ich przeprowadzania. Oczywiście logi mogą zapisywać informacje w sposób mniej lub bardziej szczegółowy – poziom szczegółowości logowania zależy od celu, w jakim chce się ich użyć. Do wykrywania błędów w funkcjonowaniu oprogramowania korzysta się zwykle z najwyższego poziomu szczegółowości.

Co mogą zawierać logi

Informacje zawarte w logu mogą dotyczyć różnych aspektów użytkowania programu

KONTROWERSJE

W przypadku różnego rodzaju komunikatorów logi mogą zawierać wykaz wiadomości oraz połączeń. Także inne informacje przechowywane i archiwizowane w formie plików log mogą nieść za sobą zagrożenia związane z ochroną prywatności. Uzyskanie dostępu do tych plików przez niepowołane osoby może doprowadzić do ujawnienia poufnych danych. Jest to jednak konsekwencja dbania o bezpieczeństwo, ponieważ brak plików log uniemożliwia analizę włamań czy awarii.

Zdarza się, że logi są dowodami przestępstw – są przekazywane organom ścigania, gdy wykazują włamania bądź też nieprawidłowe wykorzystanie systemu przez użytkowników.

– zarówno tego, co użytkownik robi z programem lokalnie, jak też tego, jakie treści są przesyłane na interfejs sieciowy. Log jest zbiorem chronologicznych wpisów. Każdy taki wpis standardowo zawiera:

- rodzaj zdarzenia, jakie wystąpiło,
- czas, w jakim doszło do zdarzenia,
- nazwę użytkownika, który wygenerował zdarzenie,
- pliki, na jakich operowało zdarzenie,
- opis zdarzenia.

Zawartość pliku log zależna jest od specyfiki oprogramowania, którego działanie log ma rejestrować.

Logback

Dla programistów pracujących w języku Java powstała biblioteka **Log4j**, która pomaga tworzyć pliki log. Wciąż jest to jedna z bibliotek najczęściej wykorzystywanych do tego celu, jednak sam projekt nie jest już rozwijany, a część osób zaangażowanych w pracę z nią rozwija teraz bibliotekę **Logback**. To ona jest obecnie rekomendowanym narzędziem do tworzenia plików log.

```
private Timer timer;
private String stanGry = "trwa";
private Logger log = LoggerFactory.getLogger(PanelGry.class);
```

1 By móc tworzyć pliki log, niezbędne jest stworzenie obiektu, który będzie za to odpowiadał. Gdybyśmy tworzyli logi do Arkanoida stworzonego w rozdziale **3**, taki obiekt można byłoby utworzyć linijką: **private Logger log = LoggerFactory.getLogger(PanelGry.class);**, gdzie **PanelGry.class** jest nazwą klasy, w której umieszczamy tworzenie logów.

2 Domyślnie zarówno **Logger**, jak i **LoggerFactory** nie będą rozpoznane przez program. By były – trzeba dodać odpowiednie importy, odpowiednio **import org.slf4j.Logger;** oraz **import org.slf4j.LoggerFactory;**.

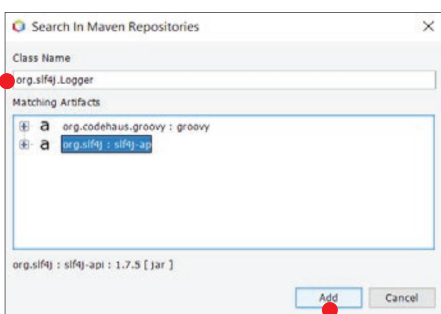
```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

3 Co istotne – importy także mogą nie być rozpoznane przez program. To dlatego,

```
import java.awt.FontMetrics;
import org.slf4j.Logger;
```
Search Dependency at Maven Repositories for org.slf4j.Logger

że te biblioteki trzeba najpierw dodać do projektu, zanim można będzie je importować. Tu z pomocą przychodzi **Maven** – klikamy na ikonę błędu przy bibliotece i wybieramy **Search Dependency at Maven Repositories for org.slf4j.Logger**.

4 Program powinien wyświetlić nowe okno, w którym powinna się znaleźć opcja **org.slf4j : slf4j-ap** – należy ją zaznaczyć i dodać do projektu, klikając na przycisk **Add**.



5 Z obiektu można korzystać, wywołując dostępne dla niego metody – jak na przykład **debug**. By rejestrować każde przesunięcie paletki w przypadku naszego Arkanoida, można dopisać wywołanie wspomnianej metody w metodzie odpowiedzial-

```
private class Sterowanie extends KeyAdapter {

    @Override
    public void keyPressed(KeyEvent e) {
        int key = e.getKeyCode();

        if (key == KeyEvent.VK_LEFT && xp > 0) {
            xp = xp - 10;
            log.debug("Paletka-strona: lewa");
        }

        if (key == KeyEvent.VK_RIGHT && xp < OKNO_SZER - szerPaletki) {
            xp = xp + 10;
            log.debug("Paletka-strona: prawa");

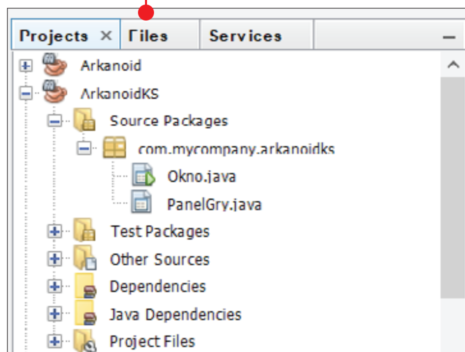
        }
    }
}
```

nej za przesuwanie paletki na boki – pisząc **log.debug("Paletka-strona: prawa");** (i analogicznie dla strony lewej).

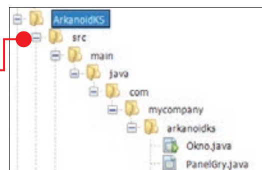
6 Po wprowadzeniu tego zapisu na standardowym wyjściu w NetBeans powinny pojawić się już wpisy zawierające informacje o ruchu paletki.

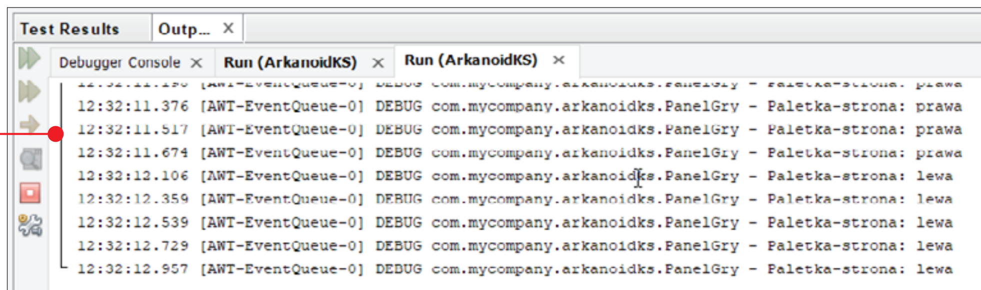
Plik konfiguracyjny

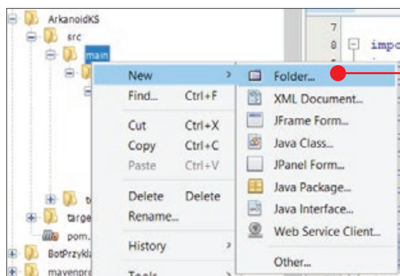
1 Domyślnie Logback kieruje wpisy na konsolę. Można jednak zmienić jego konfigurację. W tym celu dodaje się do projektu plik **logback.xml**. By go dodać, trzeba utworzyć dla niego odpowiednią lokalizację w strukturze projektu. Domyślnie projekt jest przez NetBeans wyświetlany w taki sposób, że struktura folderów nie jest widoczna. By wyświetlić projekt w formie drzewa plików, klikamy na zakładkę **Files**.



2 Rozwijamy zawartość folderu **src** z projektu, do którego chcemy wprowadzić tworzenie logów.

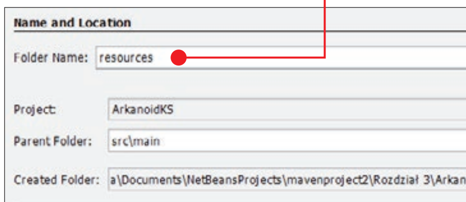




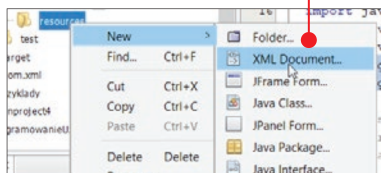


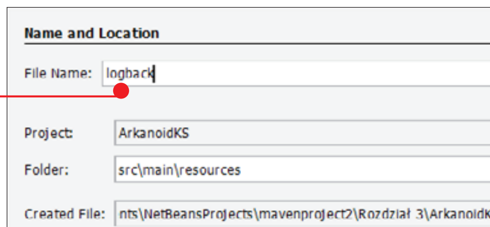
3 W folderze **main** trzeba utworzyć folder **resources** – w tym celu klikamy prawym przyciskiem myszy na folder **main** i wybieramy **New**, a następnie **Folder** ●.

4 W nowym oknie podajemy nazwę utworzonego właśnie folderu ● i zatwierdzamy przyciskiem **Finish**.



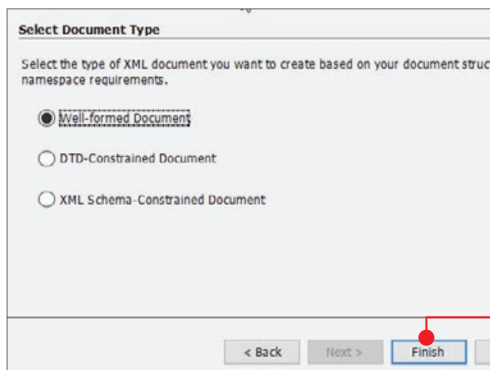
5 Klikamy prawym przyciskiem myszy na nowo utworzony folder i wybieramy z menu **New** i **XML Document** ●.





6 W nowym oknie podajemy nazwę pliku – **logback** ● – i klikamy na **Next**.

7 Pozostawiamy domyślne ustawienia i klikamy na przycisk **Finish** ●, by utworzyć dokument.



8 Przykładowy plik konfiguracyjny dla Logback może mieć następującą formę ●. To właśnie w pliku konfiguracyjnym określa się lokalizację pliku **log** czy też podstawową zawartość wpisu.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="fileAppender" class="ch.qos.logback.core.FileAppender">
        <file>c:/myLogFile.log</file>
        <append>true</append>
        <encoder>
            <pattern>%d [%thread] %-5level  %logger{35} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="TRACE">
        <appender-ref ref="fileAppender" />
    </root>
</configuration>
```

10 Dokumentacja

Dokumentacja to nie tylko instrukcje przydatne dla użytkowników, ale też dokumentacja techniczna przeznaczona dla osób, które chcą edytować program – rodzaj ściągawki ułatwiającej pracę, a także naukę kodowania

Czym jest dokumentacja

W skład dokumentacji programu wchodzi zarówno dokumentacja użytkownika, jak i dokumentacja techniczna przeznaczona dla osób mających edytować program. I właśnie tego rodzaju dokumentacją zajmiemy się w tym rozdziale. Na marginesie: Warto wiedzieć, że dokumentacja to jeden z ważnych działów inżynierii oprogramowania, dziedziny nauki porządkującej kwestie związane z tworzeniem programów.

Dokumentacja techniczna często jest wytwarzana za pomocą dodatkowego oprogramowania i w pewien sposób generowana automatycznie, by zmniejszyć nakłady pracy z nią związanej. W przypadku Javy narzędzie do generowania dokumentacji jest w pakiecie z językiem. A to oznacza, że jest standardowe – nie jest specyficzne dla wykorzystywanego IDE, ale uniwersalne. Nazywa się **Javadoc** i pozwala na generowanie dokumentacji na podstawie komentarzy. Zgodnie z powszechnie panującą opinią pisanie dokumentacji jest czynnością, którą programiści wykonują niechętnie – Javadoc znacznie przyspiesza ten proces. Korzyści z posiadania dokumentacji znacznie przewyższają nakłady czasu poniesione na jej tworzenie. Warto dbać o to, by ją na bieżąco generować.

Jak korzystać z Javadoc: dokumentacja do Arkanoida

Javadoc generuje dokumentację na podstawie komentarzy, trzeba je jednak w pewien sposób schematyzować, by program mógł je rozpoznać.
Oznaczenie komentarza, który ma być widoczny dla Javadoc, jest nieco inne od standardowego. Taki fragment kodu umieszczamy pomiędzy znacznikami **/**** – na początku, i ***/** – na końcu.

```
    /**
     * Creates new form PanelGry
     */
    public PanelGry() {
        initComponents();
        setBackground(Color.black);
        setPreferredSize(new Dimensi
```

Jeśli przyjrzymy się komentarzom, które pojawiły się w naszych skryptach wraz z generowaniem fragmentów kodu przez program, zauważymy, że część z nich jest oznaczona właśnie w ten sposób. Już na tym etapie IDE wspiera nas w tworzeniu dokumentacji.

Gdzie umieszczać komentarze

Komentarz do wygenerowania dokumentacji umieszczamy zawsze przed elementem, którego ma dotyczyć. Komentarze mogą odnosić się do klasy, metody, pakietu, pola klas – praktycznie do wszystkiego.
Dla przykładu stwórzmy dokumentację do projektu Arkanoid, który został opisany w rozdziale **3**. Zacznijmy od komentarza dotyczącego klasy – powinien on zostać już wygenerowany, a w jego treści znajduje się linijka zawierająca nazwę autora projektu. Składa się ona z parametru **@author** i nazwy użytkownika, który tworzył projekt. Zamiast tej nazwy możemy umieścić tam swoje imię i nazwisko lub pseudonim, pod jakim chcemy rozpowszechniać swoje oprogramowanie.

Co napisać w komentarzu

■ Każda linijka w komentarzu zaczyna się od znaku gwiazdki – *****. Tego też wymaga Javadoc. A jaka powinna być treść komentarza? W pierwszej linijce należy umieścić krótki opis przeznaczenia danego elementu. I tak w przypadku panelu z gry Arkanoid można napisać: **Panel służący do wyświetlania elementów gry**.

■ Dalej prócz oznaczenia autora można umieszczać też inne **tagi** – tak określamy nazwy, które w komentarzach poprzedzone są znakiem małpy, czyli **@**. W przypadku dokumentacji klasy warto użyć też tagu

```
import java.awt.FontMetrics;
/**
 *
 * @author konra
 */
public class PanelGry extends javax.swing.JPanel implements ActionListener {
```

```
import java.awt.FontMetrics;
/**
 * Panel służący do wyświetlania elementów gry.
 * @author konra
 */
public class PanelGry extends javax.swing.JPanel implements ActionListener {
```

@see, który tworzy wpis w specjalnej sekcji dokumentacji **see also**. Sekcja ta dodaje odniesienia do innych treści. Możemy tutaj wpisać dowolny tekst, na przykład tytuł książki

```
/**
 * Panel służący do wyświetlania elementów gry
 * @author konra
 * @see javax.swing.JPanel
 */
```

czy nazwę innej dokumentacji. W naszym przykładzie warto dodać odniesienie do klasy **JPanel**, poprzez dopisanie **@see javax.swing.JPanel** .

Tak jak dla całej klasy, tak i dla jej pól można tworzyć komentarze w formie Javadoc do wygenerowania dla nich dokumentacji.

```
public class PanelGry extends javax.swing
/**
 * Szerokość panelu gry
 */
    private final int OKNO_SZER = 600;
 /**
  * Wysokość panelu gry
  */
    private final int OKNO_WYS = 500;
  /**
   * Szerokość paletki do gry.
   */
    private int szerPaletki = 100;
```

W przypadku pól klasy najlepiej zapisać , do czego służy każde z nich.

Przydatne tagi

Warto zapoznać się z tagami, jakie można wykorzystać, by móc zdecydować, którego z nich użyć.

Stosując podane poniżej parametry, możemy spróbować opisać wszystkie metody z pliku **PanelGry.java**.

TAG	PARAMETRY USŁUGI
@param	Służy do opisania parametru metody. Wykorzystując go po tagu, należy wpisać nazwę parametru i jego działanie
@return	Służy do określenia, jaką wartość zwraca metoda
@since	Pozwala określić, od kiedy trwają prace nad projektem
@version	Pozwala na wprowadzenie numeracji wersji i numer wersji wyświetla w dokumentacji, gdy ta jest już wygenerowana

Dokumentacja – plik HTML

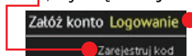
Gotowa dokumentacja jest plikiem HTML, czyli takim, który otwierany jest przez przeglądarki internetowe. Takie rozwiązanie ma sporo plusów.

Przede wszystkim można samodzielnie edytować wygląd dokumentu poprzez umieszczanie w komentarzach fragmentów skryptów HTML. Inną zaletą takiego rozwiązania jest wygodna możliwość umieszczenia takiego pliku na serwerze, by jako strona internetowa był łatwiej dostępny dla osób, które z niego korzystają.

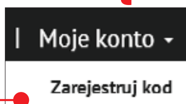
JAK SKORZYSTAĆ Z E-WYDANIA KSIĄŻKI

W KŚ+ znajdziemy **e-wydanie tej Biblioteczki, obraz ISO dołączonej do niej płyty z narzędziami dla programistów i pliki szkoleniowe do wskazówek opisanych w książce.**

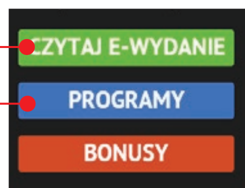
1 Otwieramy stronę **www.ksplus.pl**. Logujemy się (używamy konta z serwisu **Komputerswiat.pl**). Jeżeli nie mamy konta, klikamy na , by się zarejestrować.



2 Po zalogowaniu się możemy zarejestrować kod nadrukowany na płycie dołączonej do książki. Wystarczy kliknąć na link i przepisać kod.



3 Uzyskamy w ten sposób dostęp do e-wydania i do bonusowego obrazu płyty . Do serwisu KŚ+ możemy logować się z dowolnego urządzenia z dostępem do internetu.



UWAGA! W KŚ+ ZA DARMO E-WYDANIE KSIĄŻKI ORAZ PLIK ISO PŁYTY

POLECAMY INNE NASZE KSIĄŻKI



SPOSOBY NA HAKERÓW

Poradnik krok po kroku, jak sprawdzać bezpieczeństwo naszej sieci i urządzeń oraz blokować możliwości ataku. Na DVD Kali Linux do testów bezpieczeństwa, a w KŚ+ superpakiet do zabezpieczania Windows.



100 TRIKÓW DO ZDJĘĆ

Najlepsze triki do najlepszych darmowych programów graficznych: poprawianie i retusz, efekty i filtry, fotomontaże, własne projekty. Na DVD narzędzia pokazane we wskazówkach, a w KŚ+ – bank zdjęć.



Konrad Jagaciak
autor książki,
programista

JAVA OD PODSTAW

Java jest jednym z popularniejszych języków programowania i choć nie jest językiem najprostszym, zdecydowanie warto poświęcić czas na jego naukę. Java sprawdza się dobrze w wielu zastosowaniach. Z powodzeniem można tworzyć w tym języku nie tylko oprogramowanie użytkowe, ale też gry. Największą zaletą Javy jest wieloplatformowość. Raz napisany kod można uruchomić na wielu urządzeniach, korzystających z różnych systemów operacyjnych. Ogromną zaletą jest możliwość wykorzystania Javy do tworzenia gier i aplikacji mobilnych na urządzenia z systemem Android. Dedykowany temu program Android Studio to solidne i wygodne w pracy narzędzie.

Nauka Javy otwiera przed początkującym programistą ogrom możliwości i nowych dróg rozwoju.

W tej książce poruszono także kilka zagadnień istotnych z punktu widzenia inżynierii oprogramowania, jak testy, tworzenie dokumentacji czy pisanie logów. To tematy często pomijane na wczesnym etapie nauki programowania, a są bardzo ważne w pracy programistów.

Na płycie dołączonej do książki i w KŚ+ (www.ksplus.pl) znajdziemy zestaw najlepszych narzędzi dla programistów oraz skrypty opisane we wskazówkach.

CENA 16,90 ZŁ
W TYM 5% VAT

Płyta DVD jest dodatkiem do książki

ISBN 978-83-8091-886-3 INDEKS 321 958



Nr 2/2020 (106)



KOMPUTER
ŚWIAT
BIBLIOTECZKA